一本值得反复研读的书……

研磨设计模式

GoF设计模式细细研磨

陈臣 王斌 著

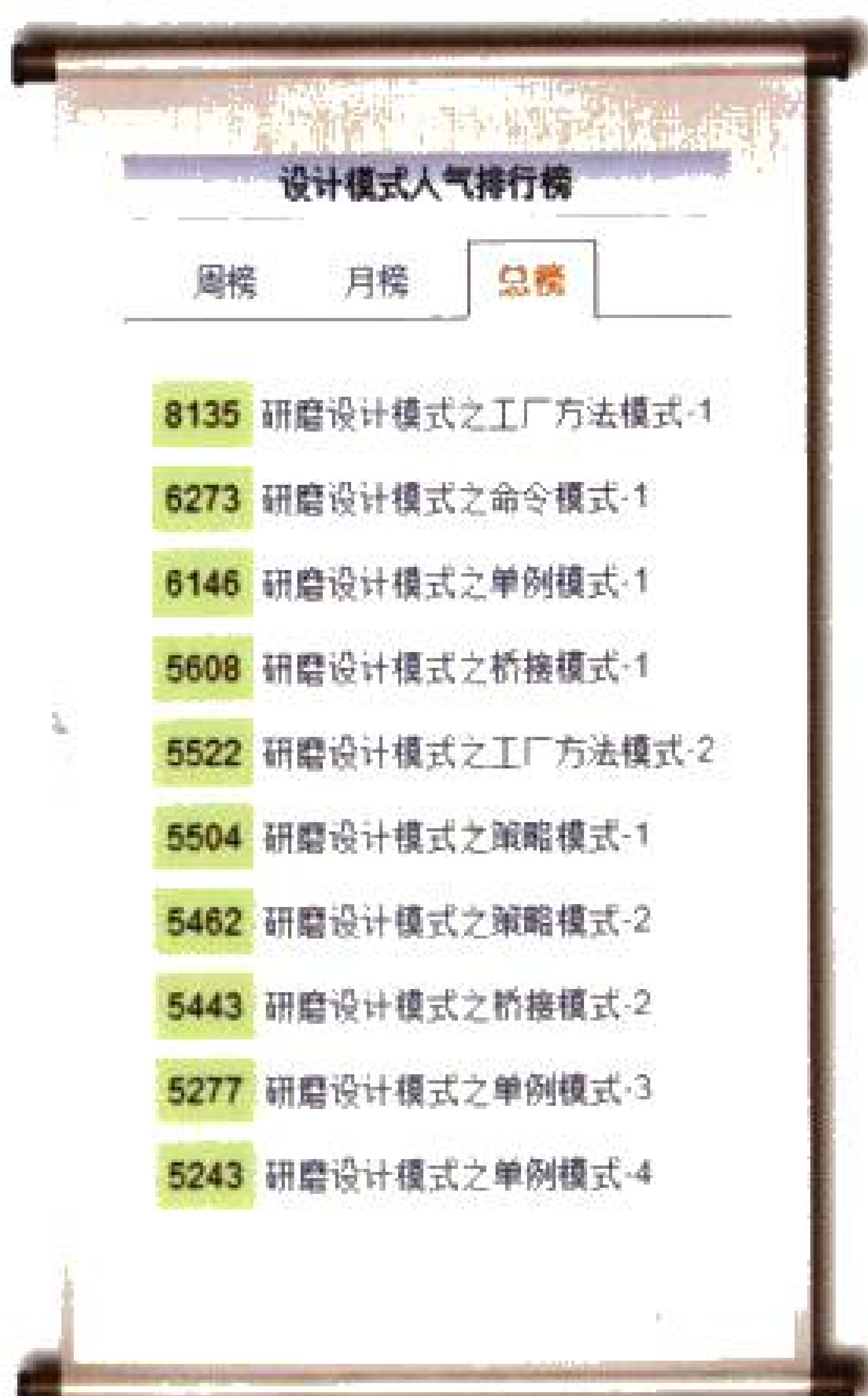


清华大学出版社

本书涉及的实际应用

- 代码生成的应用工具（独立应用）
- 日志管理平台（来自于基础平台）
- 缓存管理（来自于基础平台）
- 订单处理（来自于CRM系统）
- 导出数据的应用框架（来自于SCM）
- 组织机构管理（来自于基础平台）
- 大数据量访问（很多系统都有）
- 水质监测系统（独立应用）
- 工资管理（来自于HRM系统）
- 商品管理（来自于电子商务系统）
- 登录控制（来自于OA系统）
- 报价管理（来自于CRM系统）
- 在线投票系统（来自于OA系统）
- 仿真系统（来自于WorkFlow系统）
- 权限管理（来自于基础平台）
- 配置文件管理（来自于基础平台）
- 奖金核算系统（来自于HRM系统）
- 费用报销管理（来自于OA系统）
- 客户管理（来自于CRM系统）

JavaEye设计模式类第一名博，开博60天即垄断Javaeye年度设计模式排行TOP10。



本书涉及的实际问题

- 如何实现可配置
- 如何实现同时支持数据库和文件存储的日志管理
- 如何实现缓存以及缓存的管理
- 如何实现用缓存来控制多实例的创建
- 如何处理平行功能
- 如何实现参数化工厂
- 如何应用工厂实现DAO
- 如何实现可扩展工厂
- 如何实现原型管理器
- 如何实现Java的静态代理和动态代理
- 如何实现多线程处理队列请求
- 如何实现命令的参数化配置、可撤销的操作、宏命令、队列请求和日志请求
- 如何实现双向迭代
- 如何实现带策略的迭代器
- 如何实现翻页迭代
- 如何实现树状结构和父组件引用
- 如何检测环状结构
- 如何实现通用的增删改查
- 如何实现容错恢复机制
- 如何模拟工作流来处理流程
- 如何实现对象实例池
- 如何实现自定义语言的解析
- 如何实现简单又通用的XML读取
- 如何实现功能链，实现类似于Web开发中Filter的功能
- 如何实现模拟AOP的功能
- 如何为系统加入权限控制
- 如何自定义I/O装饰器
- 如何实现通用请求处理框架

部分评价见封底折页（根据出版规范针对个别文字进行了调整，更多中肯评价请访问：http://chjavach.javaeye.com/）

上架提示：设计模式 | JAVA

ISBN 978-7-302-23923-9



定价：89.00元

研磨【設計模式】

GoF设计模式细细研磨中……

陈臣 王斌 著

清华大学出版社
北京

内 容 简 介

本书完整覆盖GoF讲述的23个设计模式并加以细细研磨。初级内容从基本讲起，包括每个模式的定义、功能、思路、结构、基本实现、运行调用顺序、基本应用示例等，让读者能系统、完整、准确地掌握每个模式，培养正确的“设计观”；中高级内容则深入探讨如何理解这些模式，包括模式中蕴涵什么样的设计思想，模式的本质是什么，模式如何结合实际应用，模式的优缺点以及与其他模式的关系等，以期让读者尽量去理解和掌握每个设计模式的精髓所在。

本书在内容上深入、技术上实用，和实际开发结合程度很高，书中大部分的示例程序都是从实际项目中简化而来，因此很多例子都可以直接拿到实际项目中使用。如果你想要深入透彻地理解和掌握设计模式，并期望能真正把设计模式应用到项目中去，那么这是你不可错过的一本好书。

本书难度为初级到中级，适合于所有开发人员、设计人员或者即将成为开发人员的朋友。也可以作为高校学生深入学习设计模式的参考读物。

图书在版编目（CIP）数据

研磨设计模式 / 陈臣，王斌 著. —北京：清华大学出版社，2011.1

ISBN 978-7-302-23923-9

Ⅰ.①研… Ⅱ.①陈… ②王… Ⅲ. ①Java语言—程序设计 Ⅳ.①TP312

中国版本图书馆CIP数据核字(2010)第192952号

责任编辑：栾大成
责任印制：何　芊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62795954,jsjjc@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015,zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：188×260　**印　张**：50　**插　页**：1　**字　数**：1248千字
版　　次：2011年1月第1版　　**印　　次**：2011年1月第1次印刷
印　　数：1～5000
定　　价：89.00元

产品编号：039306-01

前　言

创作背景

软件开发越来越复杂，对软件设计的要求也越来越高，而软件设计和架构的入门功夫就是深入理解和掌握设计模式。因此，设计模式的重要性不言而喻。

很多朋友认识到了设计模式的重要性，也看了很多的书籍和资料，但是，常听到这样的抱怨："设计模式的书我看了不少，觉得都看懂了，就是不知道在实际开发中怎么运用这些设计模式"，从而认为设计模式是"看上去很美的花拳绣腿"。

其实不然，造成这种情况的原因就在于：这些朋友对设计模式的理解不到位，自己感觉懂了，其实还差很远，并没有"真正"理解和掌握设计模式。

市面上有不少设计模式方面的书籍，但对一般的学习者而言，要么是太深，看得云里雾里的，比如 GoF 的著作《设计模式——可复用面向对象软件的基础》，很经典，但是能吃透的人少；要么就是太浅，看了跟没看差不多，也就是介绍一下每个设计模式，告诉你这就是某某设计模式，虽然语言很生动但是实在没货，看完也不知道怎么运用，就像是带领大家摸到了设计模式的大门口，却不告诉你怎么进去一样，其根本原因还是讲得太浅，跟实际的应用有太大的差距。

对于所有想要真正理解和掌握设计模式的朋友，其实需要这样的书籍：

- 理论全面、准确，难度适中；
- 讲解深入浅出、浅显易懂；
- 理论联系实际应用，对于晦涩的理论，应有相应的示例；
- 示例最好来自实际应用，而不是来自虚拟的场景；
- 示例最好相对完整，而不是片段代码，以利于学习和应用。

这也是本书写作的目的，希望能够帮助更多的朋友早日修成设计模式的正果。

经过多年的准备和一年的写作，以及各层次读者的多轮试读意见和建议汇总，最终成书，我们可以这样说：这是一本诚意十足的书，敬请您的评鉴！

本书的试读人员包括：从还没有参加工作的学生，一直到工作 7 年的人员；职务覆盖普通的程序员、项目经理、高级系统架构师、技术部的经理；两位作者本身从事开发工作的年限，一位超过 10 年，一位超过 5 年。

试读的结果：工作经验在 1 年以下的朋友，能正常理解和掌握初级部分的内容，能部分理解中高级部分的内容；工作经验在 1~2 年的朋友，基本上能全面理解，但是领悟尚有不足；工作经验在 2~5 年的朋友，能够正常理解和掌握，基本达到本书写作的意图；工作经验在 5 年以上的朋友，主要是弥补以前较少用到的部分，使知识更加系统化和全

面化，另外把本书当作一本工具参考书，案头必备。

本书内容

本书完整覆盖 GoF 的著作《设计模式——可复用面向对象软件的基础》一书所讲述的 23 个设计模式。

- 初级内容：从基本讲起，包括每个模式的定义、功能、思路、结构、基本实现、运行调用顺序、基本应用示例等，让读者能系统、完整、准确地掌握每个模式，培养正确的“设计观”。
- 中高级内容：深入探讨如何理解这些模式、模式中蕴涵什么样的设计思想、模式的本质是什么、模式如何结合实际应用、模式的优缺点，以及和其他模式的关系等，以期让读者尽量去理解和掌握每个设计模式的精髓所在。

本书在内容上深入、技术上实用，和实际开发结合程度很高，书中大部分的示例程序都是从实际项目中简化而来，因此很多例子都可以直接拿到实际项目中使用。如果你想要深入透彻地理解和掌握设计模式，并期望能真正把设计模式应用到项目中去，那么这是你不可错过的一本好书。

本书特色

- 本书有很多独到的见解和精辟的总结，能写出一些人所不敢写、不能写的内容，是一本“真正有货”的书。
- 本书大部分示例程序都来自真实的项目应用，让你真正理解和掌握设计模式，尽量做到“从实际项目中来，再应用到实际项目中去”。
  - 本书涉及的实际应用，包含但不限于：
  - 代码生成的应用工具（独立应用）；
  - 日志管理平台（来自于基础平台）；
  - 缓存管理（来自于基础平台）；
  - 订单处理（来自于 CRM 系统）；
  - 导出数据的应用框架（来自于 SCM）；
  - 组织机构管理（来自于基础平台）；
  - 大数据量访问（很多系统都有）；
  - 水质监测系统（独立应用）；
  - 工资管理（来自于 HRM 系统）；
  - 商品管理（来自于电子商务系统）；
  - 登录控制（来自于 OA 系统）；
  - 报价管理（来自于 CRM 系统）；
  - 在线投票系统（来自于 OA 系统）；

- 仿真系统（来自于 WorkFlow 系统）；
- 权限管理（来自于基础平台）；
- 配置文件管理（来自于基础平台）；
- 奖金核算系统（来自于 HRM 系统）；
- 费用报销管理（来自于 OA 系统）；
- 客户管理（来自于 CRM 系统）。

说明：OA（Office Automation）办公自动化

CRM（Customer Relationship Management）客户关系管理

HRM（Human Resource Management）人力资源管理

SCM（Supply Chain Management) 供应链管理

WorkFlow 工作流

■ 本书探讨了很多应用设计模式来解决实际项目中的问题

本书涉及的实际问题，包含但不限于：

- 如何实现可配置；
- 如何实现同时支持数据库和文件存储的日志管理；
- 如何实现缓存以及缓存的管理；
- 如何实现用缓存来控制多实例的创建；
- 如何处理平行功能；
- 如何实现参数化工厂；
- 如何应用工厂实现 DAO；
- 如何实现可扩展工厂；
- 如何实现原型管理器；
- 如何实现 Java 的静态代理和动态代理；
- 如何实现多线程处理队列请求；
- 如何实现命令的参数化配置、可撤销的操作、宏命令、队列请求和日志请求；
- 如何实现双向迭代；
- 如何实现带策略的迭代器；
- 如何实现翻页迭代；
- 如何实现树状结构和父组件引用；
- 如何检测环状结构；
- 如何实现通用的增删改查；
- 如何实现容错恢复机制；
- 如何模拟工作流来处理流程；



- 如何实现对象实例池；
- 如何实现自定义语言的解析；
- 如何实现既简单又通用的 XML 读取；
- 如何实现功能链，实现类似于 Web 开发中 Filter 的功能；
- 如何实现模拟 AOP 的功能；
- 如何为系统加入权限控制；
- 如何自定义 I/O 装饰器；
- 如何实现通用请求处理框架。

- 本书的示例程序基本上都是带着客户端测试代码的，可直接运行，不是片段代码，更有利于大家整体学习和理解。

读者定位

本书难度为初级到中级，适合于所有开发人员、设计人员或者即将成为开发人员的朋友；也可以作为高校学生深入学习设计模式的参考读物。

我们强烈建议您认真阅读和学习本书的内容，全面、准确、深入、实用的内容定会有助于您凤凰涅磐般地实现技术升华，请相信。

阅读指南

本书假定您懂一些基本的 Java 知识，并具备一定的开发经验。

1. 对于初学设计模式的朋友

如果对常见面向对象的设计原则不太熟悉的话，请先参看附录 A。

如果对 UML 不太熟悉的话，请先参看附录 B。

然后开始看第 1 章，学习设计模式的一些基础知识，了解本书的整体大纲。

接下来就可以从前到后，循序渐进地学习每个设计模式。对每个模式建议先认真学习场景问题和解决方案两个部分，切实掌握每个模式标准的结构、实现和基本的应用。对于模式讲解中简单的内容也可以先看，但是对于后面较为复杂的内容，可以先不看，等到技术和经验积累到一定程度的时候，再循序渐进地向后学习。

2. 对于已有一定的开发经验和设计经验的朋友

首先应该从场景问题和解决方案看起，对于其中已会的内容权当复习，对于不会的内容，相当于是在查漏补缺，先把基础部分夯得全面、扎实。

然后再认真学习模式讲解部分，并结合实际的开发经验来思考，看看如何应用模式来解决实际问题、如何把模式应用到实际的项目中去，再深入地思考模式的本质和设计思想，掌握模式的精髓，这样才能真正做到在实际开发中自如地应用设计模式。

3. 对所有的朋友

这不是一本随便看看，读完一遍就可以扔掉的书籍，需要反复研读。因此，第一次阅读本书时，如果发现有些不理解的内容也不要紧，可以在今后的学习和工作中，反复参阅本书，以加深对设计模式的理解，获取设计灵感，并把设计模式切实应用到实际项目中去。

4. 善意提醒

在实际开发和设计中，要遵循简单设计的原则，不要为了模式而模式，不要过度设计，要在合适的地方应用合适的设计模式来解决问题。

这对于初学者尤其要注意，因为刚学会一个东西，总是跃跃欲试，急于一显身手，往往容易造成设计模式的误用。

本书约定

1. 本书的知识边界

由于关于设计的知识过于博大精深，因此本书"集中火力"，重点讲述 GoF 著作中涉及的 23 个设计模式本身，以及和这些设计模式相关的应用内容。

没有过多涉及：面向对象设计原则、重构、系统架构设计、JavaEE（原 J2EE，也有简写成 JEE）设计模式或是其他分类的设计模式（如 EJB 设计模式）等内容，原因可以参见附录 A。也没有过多讲述 UML，有需要的朋友请参看附录 B。

对于每章涉及的实际应用，描述也非常简略，只抽取讲述模式需要的一点东西。因为这些实际应用的东西，对于有相应开发经验的朋友多说无益，一提就明白；对于没有相应经验的朋友，多讲一点也未见得能多明白多少，反而冲淡了设计模式这个主题。

2. 本书的示例和代码

本书的示例虽然大都来自实际应用，但是经过相当的删除简化和重新组合；另外一点，为了突出设计模式这个主题，因此代码并不是按照实际应用那样来严格要求，很多例外处理、数据检测等都没有做，逻辑也未见得那么严密；还有一点，在实际的开发中，很可能是多个模式组合来实现某个功能，但是本书为了示例某个模式，让重点突出而避免读者迷惑，会选择重点示例某个模式的用法，而简化或去掉其他模式。

如果要把这些示例代码在实际应用中使用，还需要对这些代码进行加工，使其更加严谨，才能达到工业级的要求。

真诚致谢

首先要感谢清华大学出版社的员工，他们给予本书很多中肯的意见和建议，对本书从选题到出版的各个环节，都给予了大量的指导和帮助。

其次要感谢张开涛先生，他对本书的内容提出了很多有用的意见和建议。

然后要感谢魏源先生和蔡抒杨先生，他们对本书内容的完善也给出了很好的建议。

接下来，按照惯例，应该感谢家人、感谢朋友、感谢北京的漫天风沙和明媚阳光，

以及那可爱的阳台和小巧的书桌，总之，感谢一切。

最后，提前感谢购买本书的朋友们，你们的信任和赏识是我们继续前进的动力，对于本书有任何意见或建议，可以直接与我们联系，联系邮件：sjms_2010@yahoo.cn ，我们也很乐意与各位朋友交流设计模式或是其他相关的技术内容。

目　录

第 1 章　设计模式基础 1
1.1　设计模式是什么 2
1.1.1　什么是模式 2
1.1.2　设计模式的概念 2
1.1.3　设计模式的理解 3
1.1.4　设计模式的历史 4
1.2　设计模式有什么 4
1.2.1　设计模式的组成 4
1.2.2　设计模式的分类 4
1.3　设计模式的学习 5
1.3.1　为什么要学习设计模式 ..5
1.3.2　学习设计模式的层次 5
1.3.3　如何学习设计模式 6
1.4　本书的组织方式 7
1.4.1　本书所讲述的设计模式的提纲 7
1.4.2　每个模式的讲述结构 9
第 2 章　简单工厂 11
2.1　场景问题 12
2.1.1　接口回顾 12
2.1.2　面向接口编程 12
2.1.3　不用模式的解决方案 14
2.1.4　有何问题 15
2.2　解决方案 16
2.2.1　使用简单工厂来解决问题 16
2.2.2　简单工厂的结构和说明 16
2.2.3　简单工厂示例代码 17
2.2.4　使用简单工厂重写示例 19
2.3　模式讲解 20
2.3.1　典型疑问 20
2.3.2　认识简单工厂 21
2.3.3　简单工厂中方法的写法 22
2.3.4　可配置的简单工厂 24
2.3.5　简单工厂的优缺点 26
2.3.6　思考简单工厂 27
2.3.7　相关模式 27
第 3 章　外观模式 29
3.1　场景问题 30
3.1.1　生活中的示例 30
3.1.2　代码生成的应用 31
3.1.3　不用模式的解决方案 31
3.1.4　有何问题 35
3.2　解决方案 35
3.2.1　使用外观模式来解决问题 35
3.2.2　外观模式的结构和说明 36
3.2.3　外观模式示例代码 36
3.2.4　使用外观模式重写示例 39
3.3　模式讲解 40
3.3.1　认识外观模式 40
3.3.2　外观模式的实现 41
3.3.3　外观模式的优缺点 44
3.3.4　思考外观模式 44
3.3.5　相关模式 45
第 4 章　适配器模式（Adapter） 47
4.1　场景问题 48
4.1.1　装配电脑的例子 48
4.1.2　同时支持数据库和文件的日志管理 49
4.1.3　有何问题 54
4.2　解决方案 55
4.2.1　使适配器模式来解决问题 55
4.2.2　适配器模式的结构和说明 55
4.2.3　适配器模式示例代码 56
4.2.4　使用适配器模式来实现示例 58
4.3　模式讲解 61
4.3.1　认识适配器模式 61

4.3.2 适配器模式的实现 62
4.3.3 双向适配器 62
4.3.4 对象适配器和类适配器 66
4.3.5 适配器模式的优缺点 ... 69
4.3.6 思考适配器模式 70
4.3.7 相关模式 70

第5章 单例模式（Singleton）...... 73

5.1 场景问题 74
5.1.1 读取配置文件的内容 ... 74
5.1.2 不用模式的解决方案 ... 74
5.1.3 有何问题 76
5.2 解决方案 76
5.2.1 使用单例模式来解决问题 76
5.2.2 单例模式的结构和说明 77
5.2.3 单例模式示例代码 77
5.2.4 使用单例模式重写示例 80
5.3 模式讲解 82
5.3.1 认识单例模式 82
5.3.2 懒汉式和饿汉式实现 ... 83
5.3.3 延迟加载的思想 86
5.3.4 缓存的思想 87
5.3.5 Java 中缓存的基本实现 88
5.3.6 利用缓存来实现单例模式 89
5.3.7 单例模式的优缺点 90
5.3.8 在 Java 中一种更好的单例实现方式 93
5.3.9 单例和枚举 94
5.3.10 思考单例模式 95
5.3.11 相关模式 97

第6章 工厂方法模式（Factory Method）...... 99

6.1 场景问题 100
6.1.1 导出数据的应用框架 . 100
6.1.2 框架的基础知识 100
6.1.3 有何问题 102
6.2 解决方案 103
6.2.1 使用工厂方法模式来解决问题 103
6.2.2 工厂方法模式的结构和说明 104
6.2.3 工厂方法模式示例代码104
6.2.4 使用工厂方法模式来实现示例 105
6.3 模式讲解 108
6.3.1 认识工厂方法模式 108
6.3.2 工厂方法模式与 IoC/DI 112
6.3.3 平行的类层次结构 115
6.3.4 参数化工厂方法 117
6.3.5 工厂方法模式的优缺点120
6.3.6 思考工厂方法模式 121
6.3.7 相关模式 123

第7章 抽象工厂模式（Abstract Factory）...... 125

7.1 场景问题 126
7.1.1 选择组装电脑的配件 .. 126
7.1.2 不用模式的解决方案 .. 126
7.1.3 有何问题 132
7.2 解决方案 132
7.2.1 使用抽象工厂模式来解决问题 132
7.2.2 抽象工厂模式的结构和说明 133
7.2.3 抽象工厂模式示例代码 134
7.2.4 使用抽象工厂模式重写示例 136
7.3 模式讲解 140
7.3.1 认识抽象工厂模式 140
7.3.2 定义可扩展的工厂 141
7.3.3 抽象工厂模式和 DAO 146
7.3.4 抽象工厂模式的优缺点 151
7.3.5 思考抽象工厂模式 151
7.3.6 相关模式 152

第 8 章 生成器模式（Builder）........153
8.1 场景问题..............................154
8.1.1 继续导出数据的应用框架........................154
8.1.2 不用模式的解决方案..154
8.1.3 有何问题......................161
8.2 解决方案..............................161
8.2.1 使用生成器模式来解决问题..............................161
8.2.2 生成器模式的结构和说明..............................162
8.2.3 生成器模式示例代码..162
8.2.4 使用生成器模式重写示例..............................164
8.3 模式讲解..............................170
8.3.1 认识生成器模式..........170
8.3.2 生成器模式的实现......171
8.3.3 使用生成器模式构建复杂对象......................172
8.3.4 生成器模式的优点......182
8.3.5 思考生成器模式..........182
8.3.6 相关模式......................183
第 9 章 原型模式（Prototype）.. 185
9.1 场景问题..............................186
9.1.1 订单处理系统..............186
9.1.2 不用模式的解决方案..186
9.1.3 有何问题......................192
9.2 解决方案..............................193
9.2.1 使用原型模式来解决问题..............................193
9.2.2 原型模式的结构和说明..............................194
9.2.3 原型模式示例代码......194
9.2.4 使用原型模式重写示例..............................196
9.3 模式讲解..............................200
9.3.1 认识原型模式..............200
9.3.2 Java 中的克隆方法......202
9.3.3 浅度克隆和深度克隆..204
9.3.4 原型管理器..................211
9.3.5 原型模式的优缺点......214
9.3.6 思考原型模式..............215
9.3.7 相关模式......................215
第 10 章 中介者模式（Mediator）...217
10.1 场景问题............................218
10.1.1 如果没有主板............218
10.1.2 有何问题....................218
10.1.3 使用电脑来看电影....219
10.2 解决方案............................219
10.2.1 使用中介者模式来解决问题............................219
10.2.2 中介者模式的结构和说明............................220
10.2.3 中介者模式示例代码 220
10.2.4 使用中介者模式来实现示例............................223
10.3 模式讲解............................230
10.3.1 认识中介者模式........230
10.3.2 广义中介者................232
10.3.3 中介者模式的优缺点 242
10.3.4 思考中介者模式........243
10.3.5 相关模式....................243
第 11 章 代理模式（Proxy）...............245
11.1 场景问题............................246
11.1.1 访问多条数据............246
11.1.2 不用模式的解决方案 246
11.1.3 有何问题....................250
11.2 解决方案............................250
11.2.1 使用代理模式来解决问题............................250
11.2.2 代理模式的结构和说明............................251
11.2.3 代理模式示例代码....252
11.2.4 使用代理模式重写示例............................253
11.3 模式讲解............................259
11.3.1 认识代理模式............259



11.3.2 保护代理 261

11.3.3 Java 中的代理 266

11.3.4 代理模式的特点 269

11.3.5 思考代理模式 269

11.3.6 相关模式 272

第 12 章 观察者模式（Observer） .. 273

12.1 场景问题 274

12.1.1 订阅报纸的过程 274

12.1.2 订阅报纸的问题 274

12.2 解决方案 275

12.2.1 使用观察者模式来解决问题 275

12.2.2 观察者模式的结构和说明 276

12.2.3 观察者模式示例代码 277

12.2.4 使用观察者模式实现示例 279

12.3 模式讲解 283

12.3.1 认识观察者模式 283

12.3.2 推模型和拉模型 285

12.3.3 Java 中的观察者模式 289

12.3.4 观察者模式的优缺点 292

12.3.5 思考观察者模式 293

12.3.6 Swing 中的观察者模式 293

12.3.7 简单变形示例——区别对待观察者 294

12.3.8 相关模式 299

第 13 章 命令模式（Command） 301

13.1 场景问题 302

13.1.1 如何开机 302

13.1.2 与我何干 302

13.1.3 有何问题 302

13.2 解决方案 303

13.2.1 使用命令模式来解决问题 303

13.2.2 命令模式的结构和说明 304

13.2.3 命令模式示例代码 304

13.2.4 使用命令模式来实现示例 307

13.3 模式讲解 312

13.3.1 认识命令模式 312

13.3.2 参数化配置 313

13.3.3 可撤销的操作 317

13.3.4 宏命令 327

13.3.5 队列请求 333

13.3.6 日志请求 341

13.3.7 命令模式的优点 346

13.3.8 思考命令模式 347

13.3.9 退化的命令模式 347

13.3.10 相关模式 351

第 14 章 迭代器模式（Iterator） ... 353

14.1 场景问题 354

14.1.1 工资表数据的整合 354

14.1.2 有何问题 354

14.2 解决方案 354

14.2.1 使用迭代器模式来解决问题 354

14.2.2 迭代器模式的结构和说明 355

14.2.3 迭代器模式示例代码 355

14.2.4 使用迭代器模式来实现示例 359

14.3 模式讲解 368

14.3.1 认识迭代器模式 368

14.3.2 使用 Java 的迭代器 ... 370

14.3.3 带迭代策略的迭代器 373

14.3.4 双向迭代器 376

14.3.5 迭代器模式的优点 379

14.3.6 思考迭代器模式 380

14.3.7 翻页迭代 381

14.3.8 相关模式 389

第 15 章 组合模式（Composite） 391

15.1 场景问题 392

15.1.1 商品类别树 392

15.1.2 不用模式的解决方案 392
15.1.3 有何问题 395
15.2 解决方案............................396
15.2.1 使用组合模式来解决问题 396
15.2.2 组合模式的结构和说明 396
15.2.3 组合模式示例代码 397
15.2.4 使用组合模式重写示例 400
15.3 模式讲解............................405
15.3.1 认识组合模式 405
15.3.2 安全性和透明性 407
15.3.3 父组件引用 409
15.3.4 环状引用 414
15.3.5 组合模式的优缺点 418
15.3.6 思考组合模式 419
15.3.7 相关模式 419
第 16 章 模板方法模式（Template Method） 421
16.1 场景问题............................422
16.1.1 登录控制 422
16.1.2 不用模式的解决方案 422
16.1.3 有何问题 428
16.2 解决方案............................428
16.2.1 使用模板方法模式来解决问题 428
16.2.2 模板方法模式的结构和说明 429
16.2.3 模板方法模式示例代码 429
16.2.4 使用模板方法模式重写示例 430
16.3 模式讲解............................434
16.3.1 认识模板方法模式 434
16.3.2 模板的写法 438
16.3.3 Java 回调与模板方法模式 441
16.3.4 典型应用：排序 445
16.3.5 实现通用的增删改查 449
16.3.6 模板方法模式的优缺点 463
16.3.7 思考模板方法模式 463
16.3.8 相关模式 464
第 17 章 策略模式（Strategy） 465
17.1 场景问题............................466
17.1.1 报价管理 466
17.1.2 不用模式的解决方案 466
17.1.3 有何问题 467
17.2 解决方案............................469
17.2.1 使用策略模式来解决问题 469
17.2.2 策略模式的结构和说明 470
17.2.3 策略模式示例代码 470
17.2.4 使用策略模式重写示例 472
17.3 模式讲解............................475
17.3.1 认识策略模式 475
17.3.2 Context 和 Strategy 的关系 477
17.3.3 容错恢复机制 484
17.3.4 策略模式结合模板方法模式 487
17.3.5 策略模式的优缺点 490
17.3.6 思考策略模式 492
17.3.7 相关模式 493
第 18 章 状态模式（State） 495
18.1 场景问题............................496
18.1.1 实现在线投票 496
18.1.2 不用模式的解决方案 496
18.1.3 有何问题 498
18.2 解决方案............................498
18.2.1 使用状态模式来解决问题 498
18.2.2 状态模式的结构和说明 499



18.2.3 状态模式示例代码 ... 499
18.2.4 使用状态模式重写示例 ... 501
18.3 模式讲解 ... 505
18.3.1 认识状态模式 ... 505
18.3.2 状态的维护和转换控制 ... 509
18.3.3 使用数据库来维护状态 ... 514
18.3.4 模拟工作流 ... 516
18.3.5 状态模式的优缺点 ... 527
18.3.6 思考状态模式 ... 527
18.3.7 相关模式 ... 528

第 19 章 备忘录模式（Memento） ... 529

19.1 场景问题 ... 530
19.1.1 开发仿真系统 ... 530
19.1.2 不用模式的解决方案 530
19.1.3 有何问题 ... 533
19.2 解决方案 ... 533
19.2.1 使用备忘录模式来解决问题 ... 533
19.2.2 备忘录模式的结构和说明 ... 534
19.2.3 备忘录模式示例代码 535
19.2.4 使用备忘录模式重写示例 ... 537
19.3 模式讲解 ... 541
19.3.1 认识备忘录模式 ... 541
19.3.2 结合原型模式 ... 544
19.3.3 离线存储 ... 546
19.3.4 再次实现可撤销操作 549
19.3.5 备忘录模式的优缺点 558
19.3.6 思考备忘录模式 ... 558
19.3.7 相关模式 ... 559

第 20 章 享元模式（Flyweight） . 561

20.1 场景问题 ... 562
20.1.1 加入权限控制 ... 562
20.1.2 不使用模式的解决方案 ... 563
20.1.3 有何问题 ... 568
20.2 解决方案 ... 569
20.2.1 使用享元模式来解决问题 ... 569
20.2.2 享元模式的结构和说明 ... 570
20.2.3 享元模式示例代码 ... 570
20.2.4 使用享元模式重写示例 ... 573
20.3 模式讲解 ... 578
20.3.1 认识享元模式 ... 578
20.3.2 不需要共享的享元实现 ... 580
20.3.3 对享元对象的管理 ... 587
20.3.4 享元模式的优缺点 ... 596
20.3.5 思考享元模式 ... 597
20.3.6 相关模式 ... 597

第 21 章 解释器模式（Interpreter） ... 599

21.1 场景问题 ... 600
21.1.1 读取配置文件 ... 600
21.1.2 不用模式的解决方案 600
21.1.3 有何问题 ... 602
21.2 解决方案 ... 604
21.2.1 使用解释器模式来解决问题 ... 604
21.2.2 解释器模式的结构和说明 ... 605
21.2.3 解释器模式示例代码 605
21.2.4 使用解释器模式重写示例 ... 607
21.3 模式讲解 ... 615
21.3.1 认识解释器模式 ... 615
21.3.2 读取多个元素或属性的值 ... 617
21.3.3 解析器 ... 625
21.3.4 解释器模式的优缺点 633
21.3.5 思考解释器模式 ... 633

21.3.6 相关模式....................634

第 22 章 装饰模式（Decorator）.....635

22.1 场景问题............................636
22.1.1 复杂的奖金计算........636
22.1.2 简化后的奖金计算体系...........................636
22.1.3 不用模式的解决方案 636
22.1.4 有何问题....................639
22.2 解决方案............................640
22.2.1 使用装饰模式来解决问题............................640
22.2.2 装饰模式的结构和说明............................641
22.2.3 装饰模式示例代码....642
22.2.4 使用装饰模式重写示例............................644
22.3 模式讲解............................650
22.3.1 认识装饰模式...........650
22.3.2 Java 中的装饰模式应用...........................653
22.3.3 装饰模式和 AOP.......657
22.3.4 装饰模式的优缺点....664
22.3.5 思考装饰模式...........664
22.3.6 相关模式....................665

第 23 章 职责链模式（Chain of Responsibility）................667

23.1 场景问题............................668
23.1.1 申请聚餐费用...........668
23.1.2 不用模式的解决方案 668
23.1.3 有何问题....................671
23.2 解决方案............................671
23.2.1 使用职责链模式来解决问题............................671
23.2.2 职责链模式的结构和说明............................672
23.2.3 职责链模式示例代码 673
23.2.4 使用职责链模式重写示例............................674
23.3 模式讲解............................679
23.3.1 认识职责链模式........679
23.3.2 处理多种请求............680
23.3.3 功能链........................691
23.3.4 职责链模式的优缺点 697
23.3.5 思考职责链模式........697
23.3.6 相关模式....................698

第 24 章 桥接模式（Bridge）............701

24.1 场景问题............................702
24.1.1 发送提示消息............702
24.1.2 不用模式的解决方案 702
24.1.3 有何问题....................705
24.2 解决方案............................707
24.2.1 使用桥接模式来解决问题............................707
24.2.2 桥接模式的结构和说明............................708
24.2.3 桥接模式示例代码....709
24.2.4 使用桥接模式重写示例............................710
24.3 模式讲解............................715
24.3.1 认识桥接模式............715
24.3.2 谁来桥接....................718
24.3.3 典型例子——JDBC...721
24.3.4 广义桥接——Java 中无处不桥接..................723
24.3.5 桥接模式的优点........726
24.3.6 思考桥接模式............727
24.3.7 相关模式....................728

第 25 章 访问者模式（Visitor）......731

25.1 场景问题............................732
25.1.1 扩展客户管理的功能 732
25.1.2 不用模式的解决方案 734
25.1.3 有何问题....................738
25.2 解决方案............................739
25.2.1 使用访问者模式来解决问题............................739

25.2.2 访问者模式的结构和说明 739
25.2.3 访问者模式示例代码 740
25.2.4 使用访问者模式重写示例 744
25.3 模式讲解 749
25.3.1 认识访问者模式 749
25.3.2 操作组合对象结构 ... 751
25.3.3 谁负责遍历所有元素对象 758
25.3.4 访问者模式的优缺点 761
25.3.5 思考访问者模式 761
25.3.6 相关模式 762
附录 A 常见面向对象设计原则 763
A.1 设计模式和设计原则 764
A.1.1 设计模式和设计原则的关系 764
A.1.2 为何不重点讲解设计原则 764
A.2 常见的面向对象设计原则 . 765
A.2.1 单一职责原则 SRP（Single Responsibility Principle） 765
A.2.2 开放-关闭原则 OCP（Open-Closed Principle）765
A.2.3 里氏替换原则 LSP（Liskov Substitution Principle） 765
A.2.4 依赖倒置原则 DIP（Dependence Inversion Principle） 766
A.2.5 接口隔离原则 ISP（Interface Segregation Principle） 766
A.2.6 最少知识原则 LKP（Least Knowledge Principle） 767
A.2.7 其他原则..................... 767
附录 B UML 简介 769
B.1 UML 基础........................... 770
B.1.1 UML 是什么 770
B.1.2 UML 历史 770
B.1.3 UML 能干什么 771
B.1.4 UML 有什么 771
B.2 类图.................................... 772
B.2.1 类图的概念 772
B.2.2 类图的基本表达 772
B.2.3 抽象类和接口 773
B.2.4 关系 774
B.3 顺序图................................. 778
B.3.1 顺序图的概念 778
B.3.2 顺序图的基本表达 779
临别赠言 782
不是结束而是新的开始.......... 782
你该怎么做.............................. 782
参考文献 783

第1章　设计模式基础

1.1 设计模式是什么

1.1.1 什么是模式

从字面上理解，模，就是模型、模板的意思；式，就是方式、方法的意思。综合起来，所谓模式就是：可以作为模型或模板的方式或方法。再简单点说就是可以用来作为样板的方式或方法，类似于大家所熟悉的范例。

1.1.2 设计模式的概念

按照上面的理解，设计模式指的就是设计方面的模板，也即设计方面的方式或方法。

> 设计模式：是指在软件开发中，经过验证的，用于解决在特定环境下、重复出现的、特定问题的解决方案。

1. 设计模式是**解决方案**

根据上面对设计模式的定义可以看出，归根结底，设计模式就是一些解决方案。

所谓解决方案，就是解决办法，亦即是解决问题的方式或方法。通常所说的方案书，就是把解决方法文档化后形成的文档。

那么，能不能反过来说：解决方案就是设计模式呢？很明显是不行的。为什么呢？因为在解决方案之前还有一些定语，只有满足这些条件的解决方案才被称为设计模式。

下面就一个个来看。

2. 设计模式是**特定问题**的解决方案

为什么要限制设计模式是“特定问题”的解决方案呢？

限制“特定问题”，说明设计模式不是什么万能灵药，并不是什么问题都能解决，通常一个设计模式仅仅解决某个或某些特定的问题，并不能包治百病。

因此不要迷信设计模式，也不要泛滥使用设计模式，设计模式解决不了那么多问题，它只是解决“特定问题”的解决方案。

3. 设计模式是**重复出现**的、特定问题的解决方案

那么为何要这些特定问题是“重复出现”的呢？

只有这些特定问题“重复出现”，那么为这些问题总结解决方案才是有意义的行为。因为只有总结了这些问题的解决方案，当这些问题再次出现的时候，就可以复用这些解决方案，而不用从头来寻求解决办法了。

4. 设计模式是用于解决在**特定环境下**、重复出现的、特定问题的解决方案

为什么要限制在“特定环境下”呢？

任何问题的出现都是有场景的，不能脱离环境去讨论对问题的解决办法，因为不同

环境下，就算是相同的问题，解决办法也不一定是一样的。

5. 设计模式是**经过验证**的，用于解决在特定环境下、重复出现的、特定问题的解决方案，为什么要限制是“经过验证的”呢？

每个人都可以总结一些用于解决在特定环境下、重复出现的、特定问题的解决方案，但并不是每个人总结的解决方案都算得上是设计模式，这些解决方案应该要有足够的应用来验证，并得到大家的认可和公认。只有经过验证的解决方案才算得上是设计模式。

没有得到验证的解决方案，假如也算设计模式而被大家大量复用的话，万一这个方案有问题呢？那么所有应用它的地方都会出错，都应该修改，这种复用还不如不用呢。

6. 为何要强调“在软件开发中”

原因很简单，因为接下来要讨论的内容，就是软件开发中的设计模式，因此这里限制“在软件开发中”。

注意　**并不能说设计模式是软件行业独有的，事实上，很多行业都有自己的设计模式。**

1.1.3　设计模式的理解

通过上面对设计模式概念的讲述，可以看出，设计模式也没有什么神奇之处，下面对设计模式再做几点说明，使读者进一步理解它。

- 设计模式是解决某些问题的办法。

  要理解和掌握设计模式，其重心就在于对这些办法的理解和掌握，然后进一步深化到这些办法所体现的思想层面上，将设计模式所体现的思考方式进行吸收和消化，融入到自己的思维中。

- 设计模式不是凭空想象出来的，是经验的积累和总结。

  从理论上来说，设计模式并不一定是最优秀的解决方案，有可能存在比设计模式更优秀的解决方案，也就是说设计模式是相对优秀的，没有最优，只有更优。

延伸　**这也说明，从理论上，我们自己也可以总结一些这样的解决方案，如果能得到大家的认可和验证，也是有可能成为公认的设计模式的。**

- 设计模式并不是一成不变的，而是在不断发展中。

  本书仅仅讨论 GoF 的著作中所记载的、经典的设计模式，但并不是说只有这些设计模式。因为设计模式的发展从设计模式引入软件中以来，就从来没有停止过。

- 设计模式并不是软件行业独有的，各行各业都有自己的设计模式。

  用大家身边的例子来说，比如医药行业，就有自己的设计模式。假设一个人感冒了，到药店买感冒药，这个感冒药就是设计模式的一个很好体现。

  - 经过验证的：药品上市前，会有大量的验证和实验，以保证药品的安全性。
  - 特定环境下：这些药品是针对人的，不是针对其他动物的。
  - 重复出现的：正是因为感冒会重复出现，研制药品才是有意义的。

- 结构型模式：描述如何组合类和对象以获得更大的结构。
- 行为型模式：描述算法和对象间职责的分配。

当然也有按其他方式进行分类的，这里就不再讨论了。

1.3　设计模式的学习

1.3.1　为什么要学习设计模式

为什么要学习设计模式？实在是有太多的理由了，这里简单地罗列几点。

1. 设计模式已经成为软件开发人员的“标准词汇”

很多软件开发人员在相互交流的时候，只是使用设计模式的名称，而不深入说明其具体内容。就如同我们在汉语里面使用成语一样，当你在交流中使用一个成语的时候，是不会去讲述这个成语背后的故事的。

举个例子来说：开发人员 A 碰到了一个问题，然后与开发人员 B 讨论，开发人员 B 可能会支招：使用“XXX 模式”（XXX 是某个设计模式的名称）就可以了。如果这个时候开发人员 A 不懂设计模式，那他们就无法交流。

因此，一个合格的软件开发人员，必须掌握设计模式这个“标准词汇”。

2. 学习设计模式是个人技术能力提高的捷径

设计模式是很多前辈经验的积累，大都是一些相对优秀的解决方案，很多问题都是典型的、有代表性的问题。

学习设计模式，可以学习到众多前辈的经验，吸收和领会他们的设计思想，掌握他们解决问题的方法，就相当于站在这些巨人的肩膀上，可以让我们个人的技术能力得到快速的提升。学习设计模式虽然有一定的困难，但绝对是快速提高个人技术能力的捷径。

3. 不用重复发明轮子

设计模式是解决某些特定问题的解决方案。当我们再次面对这些问题的时候，就不用自己从头来解决这些问题，复用这些方案即可。

大多数情况下，这或许是比自己从头来解决这些问题更好的方案。一是你未必能找到比设计模式更优秀的解决方案；另外，通过使用设计模式可以节省大量的时间，你可以把节省的时间花在其他更需要解决的问题上。

1.3.2　学习设计模式的层次

学习设计模式大致有以下三个层次。

1. 基本入门级

要求能够正确理解和掌握每个设计模式的基本知识，能够识别在什么场景下、出现了什么样的问题、采用何种方案来解决它，并能够在实际的程序设计和开发中套用相应的设计模式。

2. 基本掌握级

除了具备基本入门级的要求外，还要求能够结合实际应用的场景，对设计模式进行变形使用。

事实上，在实际开发中，经常会碰到与标准模式的应用场景有一些不一样的情况，此时要合理地使用设计模式，就需要对它们做适当的变形，而不是僵硬地套用了。当然进行变形的前提是要能准确深入地理解和把握设计模式的本质，**万变不离其宗，只有把握住本质，才能够确保正确变形使用而不是误用。**

3. 深入理解和掌握级

除了具备基本掌握级的要求外，更主要的是：

> 延伸　**要从思想上和方法上吸收设计模式的精髓，并融入到自己的思路中，在进行软件的分析和设计的时候，能随意地、自然而然地应用，就如同自己思维的一部分。**

在较复杂的应用中，当解决某个问题的时候，很可能不是单一应用某一个设计模式，而是综合应用很多设计模式。例如，结合某个具体的情况，可能需要把模式 A 进行简化，然后结合模式 B 的一部分，再组合应用变形的模式 C…，如此来解决实际问题。

更复杂的是除了考虑这些设计模式外，还可能需要考虑系统整体的体系结构、实际功能的实现、与已有功能的结合等。这就要求在应用设计模式的时候，不拘泥于设计模式本身，而是从思想和方法的层面进行应用。

简单点说，基本入门级就是套用使用，相当于能够依葫芦画瓢，很机械；基本掌握级就是能变形使用，比基本入门级灵活一些，可以适当变形使用；深入理解和掌握级才算是真正将设计模式的精髓吸收了，是从思想和方法的层面去理解和掌握设计模式，就犹如练习武功到最高境界，“无招胜有招”了。要想达到这个境界，没有足够的开发和设计经验，没有足够深入的思考，是不太可能达到的。

> 提示　**有些朋友说：设计模式的书我看了不少，觉得都看懂了，就是不知道在实际开发中怎么用这些设计模式，于是他们认为设计模式是“看上去很美”的“花拳绣腿”。其实这些朋友正处于“设计模式了解级”，根本还没有入门。**

1.3.3　如何学习设计模式

结合作者自身的经验，给出以下学习设计模式的建议。

（1）首先要调整好心态，不要指望一蹴而就，不可浮躁。

学习和掌握设计模式需要一个过程，不同的阶段看这些设计模式会有不同的领悟和感受。

不要指望真正的设计模式的书籍是既简单又有趣的，一看就懂的。**那种书籍多是属于科普性质的书籍**，只是让你简单了解一下设计模式。这也是为何很多朋友总感觉“懂”设计模式，却不会在实际项目中应用设计模式。那是因为你“懂”的程度不够。

提示	**要想真正理解和掌握，必须要上升到一定的难度和深度，让你看完后思考，思考后应用，然后再看、再思考、再应用，如此反复，方能成就。** **“鱼和熊掌不可兼得”，因此，本书尽量在内容的深度、难度和讲述的通俗易懂、简单明了上进行均衡，以期大家能以较小的力气去真正理解和掌握设计模式。**

（2）学习设计模式的第一步：准确理解每个设计模式的功能、基本结构、标准实现，了解适合使用它的场景以及使用的效果

（3）学习设计模式的第二步：实际的开发中，尝试着使用这些设计模式，并反复思考和总结是否使用得当，是否需要做一些变化。

（4）学习设计模式的第三步：再回头去看设计模式的理论，有了实际的模式应用经验再看设计模式，会有不同的感悟，一边看一边结合着应用经验来思考。比如：设计模式的本质功能是什么？它是如何实现的？这种实现方式还可以在什么地方应用？如何才能把这个设计模式和具体的应用结合起来？这个设计模式设计的出发点是什么？等等。可以有很多考虑的点，从不同的角度对设计模式进行思考。

（5）第四步：多次重复学习设计模式的第二步和第三步。也就是在实际开发中使用，然后结合理论思考，然后再应用，再思考……如此循环，反复多次，直到达到对设计模式基本掌握的水平。

简而言之，大家要注意使设计模式的理论和实践相结合，**理论指导实践，实践反过来加深对理论的理解**，如此反复循环，成螺旋式上升。

事实上，到了基本掌握设计模式的水平后，最后能达到一个什么样的高度，因人而异，需要看个人的思维水平和理解水平。对于这个阶段，只有一个建议，那就是反复地、深入地思考，别无他法。到了思想的层面，就得靠“悟”了。

1.4　本书的组织方式

1.4.1　本书所讲述的设计模式的提纲

从第 3 章开始，本书详细地讲述了《设计模式——可复用面向对象软件的基础》（GoF 著）一书所讲述的 23 个设计模式；第 2 章讲述的简单工厂，严格意义上并不算是标准的设计模式，只能算是一个热身运动。

1. 第 2 章　简单工厂（GoF 的著作中没有）

提供一个创建对象实例的功能，而无须关心其具体实现。被创建实例的类型可以是

接口、抽象类，也可以是具体的类。

2. 第 3 章　外观模式（GoF 的著作中划分为结构型）

为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

3. 第 4 章　适配器模式（GoF 的著作中划分为结构型）

将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

4. 第 5 章　单例模式（GoF 的著作中划分为创建型）

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

5. 第 6 章　工厂方法模式（GoF 的著作中划分为创建型）

定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method 使一个类的实例化延迟到其子类。

6. 第 7 章　抽象工厂模式（GoF 的著作中划分为创建型）

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

7. 第 8 章　生成器模式（GoF 的著作中划分为创建型）

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

8. 第 9 章　原型模式（GoF 的著作中划分为创建型）

用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

9. 第 10 章　中介者模式（GoF 的著作中划分为行为型）

用一个中介对象来封装一系列的对象交互。中介者使得各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

10. 第 11 章　代理模式（GoF 的著作中划分为结构型）

为其他对象提供一种代理以控制对这个对象的访问。

11. 第 12 章　观察者模式（GoF 的著作中划分为行为型）

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

12. 第 13 章　命令模式（GoF 的著作中划分为行为型）

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。

13. 第 14 章　迭代器模式（GoF 的著作中划分为行为型）

提供一种方法顺序访问一个聚合对象中的各个元素，而又不需暴露该对象的内部表示。

14. 第 15 章　组合模式（GoF 的著作中划分为结构型）

将对象组合成树形结构以表示“部分—整体”的层次结构。组合模式使得用户对单

个对象和组合对象的使用具有一致性。

15. 第 16 章　模板方法模式（GoF 的著作中划分为行为型）

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

16. 第 17 章　策略模式（GoF 的著作中划分为行为型）

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

17. 第 18 章　状态模式（GoF 的著作中划分为行为型）

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

18. 第 19 章　备忘录模式（GoF 的著作中划分为行为型）

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样，以后就可将该对象恢复到原先保存的状态。

19. 第 20 章　享元模式（GoF 的著作中划分为结构型）

运用共享技术有效地支持大量细粒度的对象。

20. 第 21 章　解释器模式（GoF 的著作中划分为行为型）

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

21. 第 22 章　装饰模式（GoF 的著作中划分为结构型）

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式比生成子类更为灵活。

22. 第 23 章　职责链模式（GoF 的著作中划分为行为型）

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

23. 第 24 章　桥接模式（GoF 的著作中划分为结构型）

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

24. 第 25 章　访问者模式（GoF 的著作中划分为行为型）

表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

1.4.2　每个模式的讲述结构

1. 场景问题

1）某个实际应用

通过一个实际的应用来描述在某个场景下出现的某个问题，也就是模式要解决的问

题。

2）不用模式的解决方案

先来看下不用模式，如何来解决这个问题。

3）有何问题

分析不用模式的解决方案中存在的问题，指出需要寻找更好的解决方案。

2. 解决方案

1）某个模式来解决

首先是模式的定义，然后描述应用这个模式来解决上面提出的问题的解决思路。

2）模式结构和说明

使用 UML 画出模式的结构图，并说明各个参与者。

3）模式的示例代码

尽量准确地给出每个模式的基本实现的示例代码，算是模式的一个标准参考实现。

4）使用模式来重写示例

使用模式来重写前面“不用模式”所实现的示例，既作为模式的一个实际应用示例，也方便大家对比学习和体会，看看如何使用模式来解决问题，以及使用模式的好处。

3. 模式讲解

1）认识某个模式

主要是对模式所涉及的各种知识进行讲述，通常会包含模式的功能、对各部分的理解、对模式实现的探讨、模式运行的顺序等等。

2）针对各个重点难点功能，或是与实际应用结合的讨论和示例

针对模式的一些重点难点，或是模式与实际应用结合，来进行深入的讲解和示例。

3）模式的优缺点

讨论模式的优缺点，可以使你在实际应用中尽量使用模式的优点，而规避使用模式的缺点。

4）思考模式

先总结模式的本质，再从设计上思考模式，然后给出适合使用模式的情况。

5）相关模式

描述与其他模式的关系，以及与其他模式相比较的异同点。

第2章 简单工厂

简单工厂不是一个标准的设计模式，但是它实在是太常用了，简单而又神奇，所以需要好好掌握它，就当是学习设计模式的热身运动吧。

为了保持一致性，我们尽量按照学习其他模式的步骤来进行学习。

2.1 场景问题

大家都知道，在 Java 应用开发中，要“面向接口编程”。

那么什么是接口？接口有什么作用？接口如何使用？我们一起来回顾一下。

2.1.1 接口回顾

1. Java 中接口的概念

在 Java 中接口是一种特殊的抽象类，跟一般的抽象类相比，接口里面的所有方法都是抽象方法，接口里面的所有属性都是常量。也就是说，接口里面只有方法定义而没有任何方法实现。

2. 接口用来干什么

通常用接口来定义实现类的外观，也就是实现类的行为定义，用来约束实现类的行为。接口就相当于一份契约，根据外部应用需要的功能，约定了实现类应该要实现的功能，但是具体的实现类除了实现接口约定的功能外，还可以根据需要实现其他一些功能，这是允许的，也就是说实现类的功能包含但不仅限于接口约束的功能。

通过使用接口，可以实现不相关类的相同行为，而不需考虑这些类之间的层次关系，接口就是实现类对外的外观。

3. 接口的思想

根据接口的作用和用途，浓缩下来，**接口的思想就是“封装隔离”**。

通常提到的封装是指对数据的封装，但是这里的封装是指“对被隔离体的行为的封装”，或者是“对被隔离体的职责的封装”；而隔离指的是外部调用和内部实现，外部调用只能通过接口进行调用，外部调用是不知道内部具体实现的，也就是说外部调用和内部实现是被接口隔离开的。

4. 使用接口的好处

由于外部调用和内部实现被接口隔离开了，那么只要接口不变，内部实现的变化就不会影响到外部应用，从而使得系统更灵活，具有更好的扩展性和可维护性，这也就是所谓“接口是系统可插拔性的保证”这句话的意思。

5. 接口和抽象类的选择

既然接口是一种特殊的抽象类，那么在开发中，何时选用接口？何时选用抽象类呢？

对于它们的选择，在开发中是一个很重要的问题，特别总结两句话给大家：

- 优先选用接口
- 在既要定义子类的行为，又要为子类提供公共的功能时应选择抽象类。

2.1.2 面向接口编程

面向接口编程是 Java 编程中的一个重要原则。

在 Java 程序设计里面，非常讲究层的划分和模块的划分。通常按照三层来划分 Java 程序，分别是表现层、逻辑层和数据层，它们之间都要通过接口来通信。

在每一个层里面，又有很多个小模块，每个小模块对外则是一个整体，所以一个模块对外应该提供接口，其他地方需要使用到这个模块的功能时，可以通过此接口来进行调用。这也就是常说的“接口是被其隔离部分的外观”。基本的三层结构如图 2.1 所示。

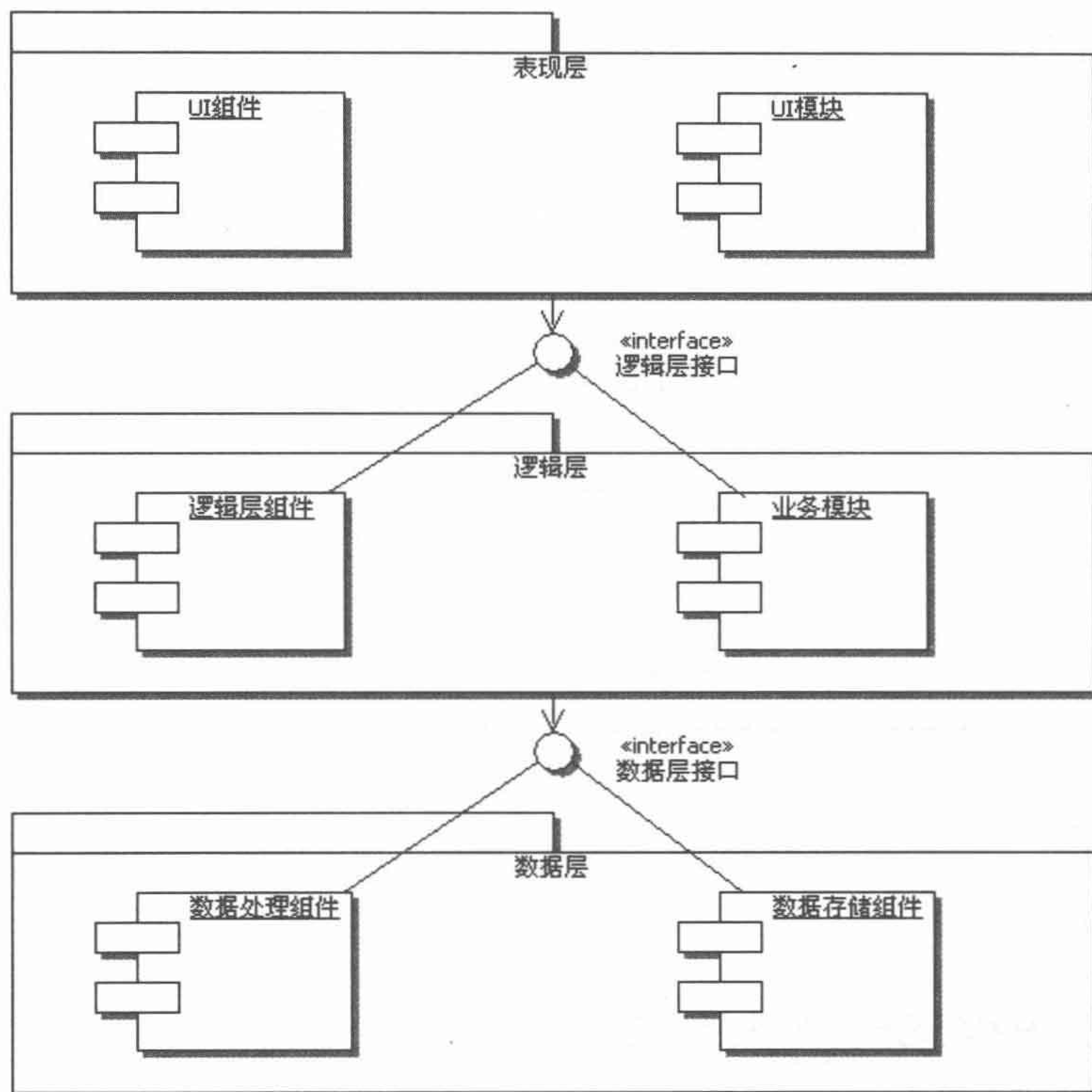


图 2.1　基本的三层结构示意图

在一个层内部的各个模块间的交互要通过接口，如图 2.2 所示。

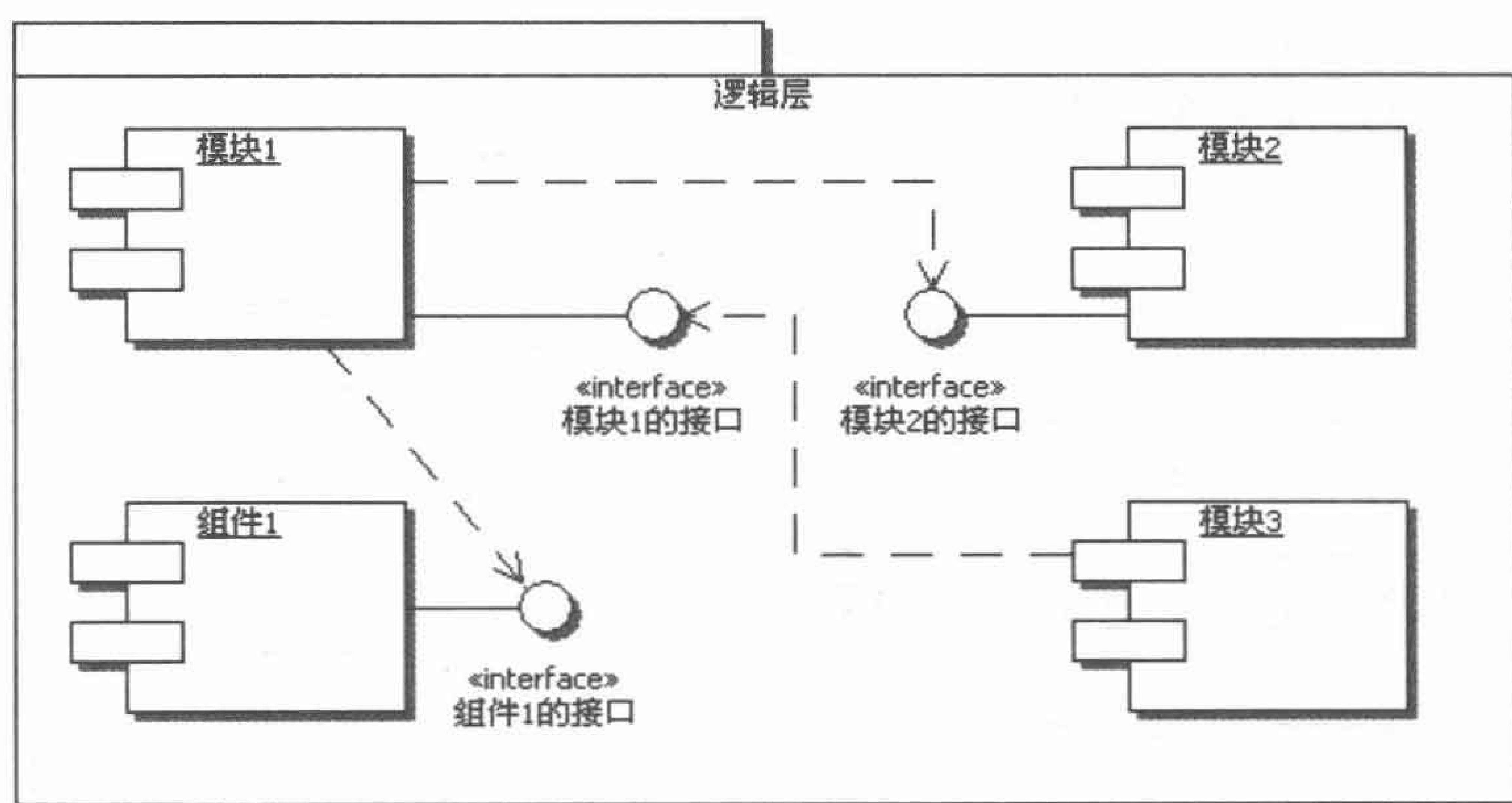


图 2.2　一个层内部的各个模块间的交互示意图

各个部分的接口具体应该如何去定义，具体的内容是什么，我们不去深究，那是需要具体问题具体分析的，这里仅学习设计的方法。

上面频频提到“组件”，那么什么是组件呢？

所谓组件：**从设计上讲，组件就是能完成一定功能的封装体。**小到一个类，大到一个系统，都可以称为组件，因为一个小系统放到更大的系统里面去，也就当个组件而已。事实上，从设计的角度看，系统、子系统、模块、组件等说的其实是同一回事情，都是完成一定功能的封装体，只不过功能多少不同而已。

继续刚才的思路，大家会发现，不管是一层还是一个模块或者一个组件，都是一个被接口隔离的整体，那么下面我们就不去区分它们，统一认为它们都是接口隔离体即可，如图 2.3 所示。

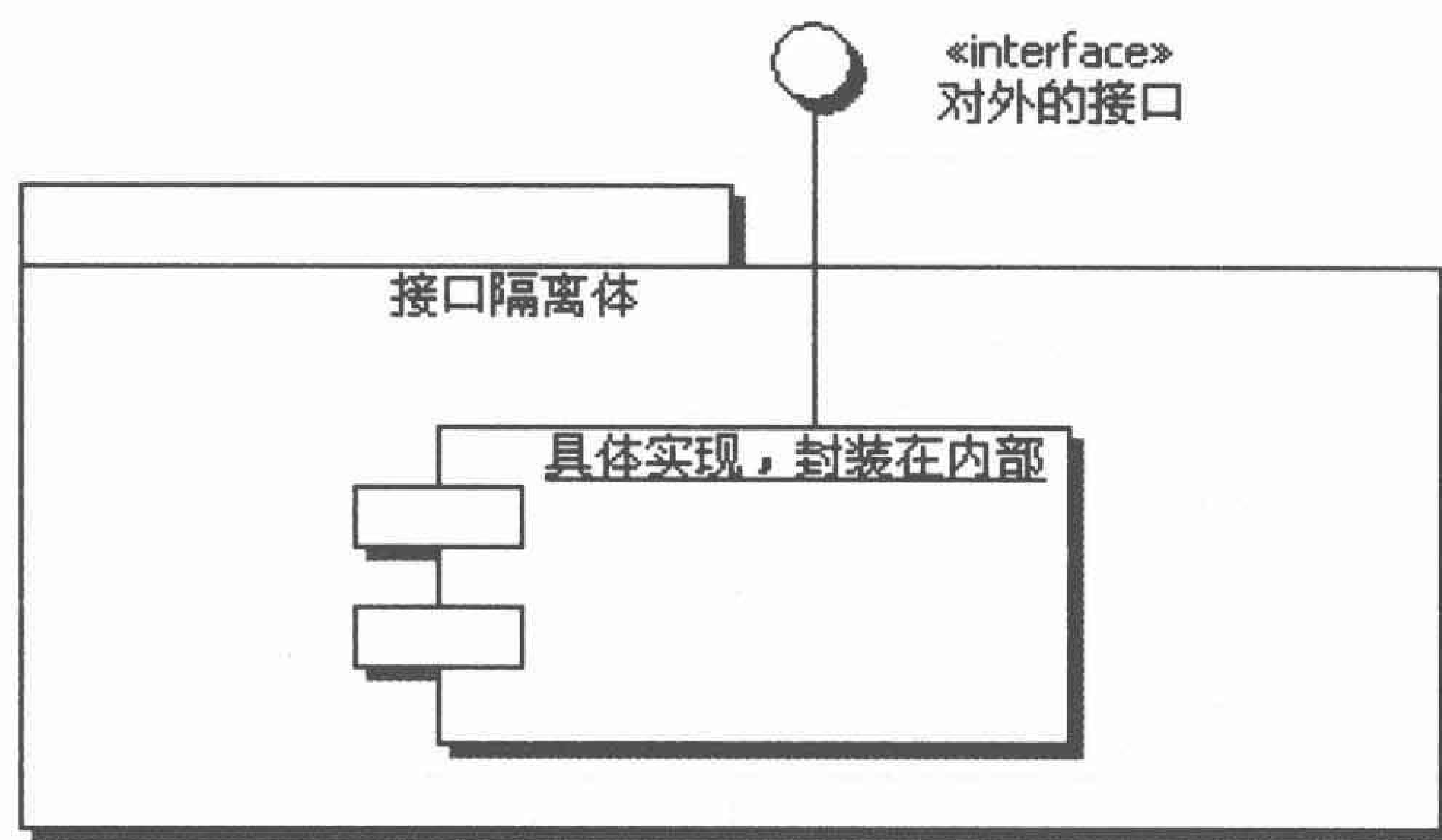


图 2.3　接口隔离体示意图

既然在 Java 中需要面向接口编程，那么在程序中到底如何使用接口，来做到真正的面向接口编程呢？

2.1.3　不用模式的解决方案

回忆一下，以前是如何使用接口的呢，假设有一个接口叫 Api，然后有一个实现类 Impl 实现了它，在客户端怎么用这个接口呢？

通常都是在客户端创建一个 Impl 的实例，把它赋值给一个 Api 接口类型的变量，然后客户端就可以通过这个变量来操作接口的功能了，此时具体的结构图如图 2.4 所示。

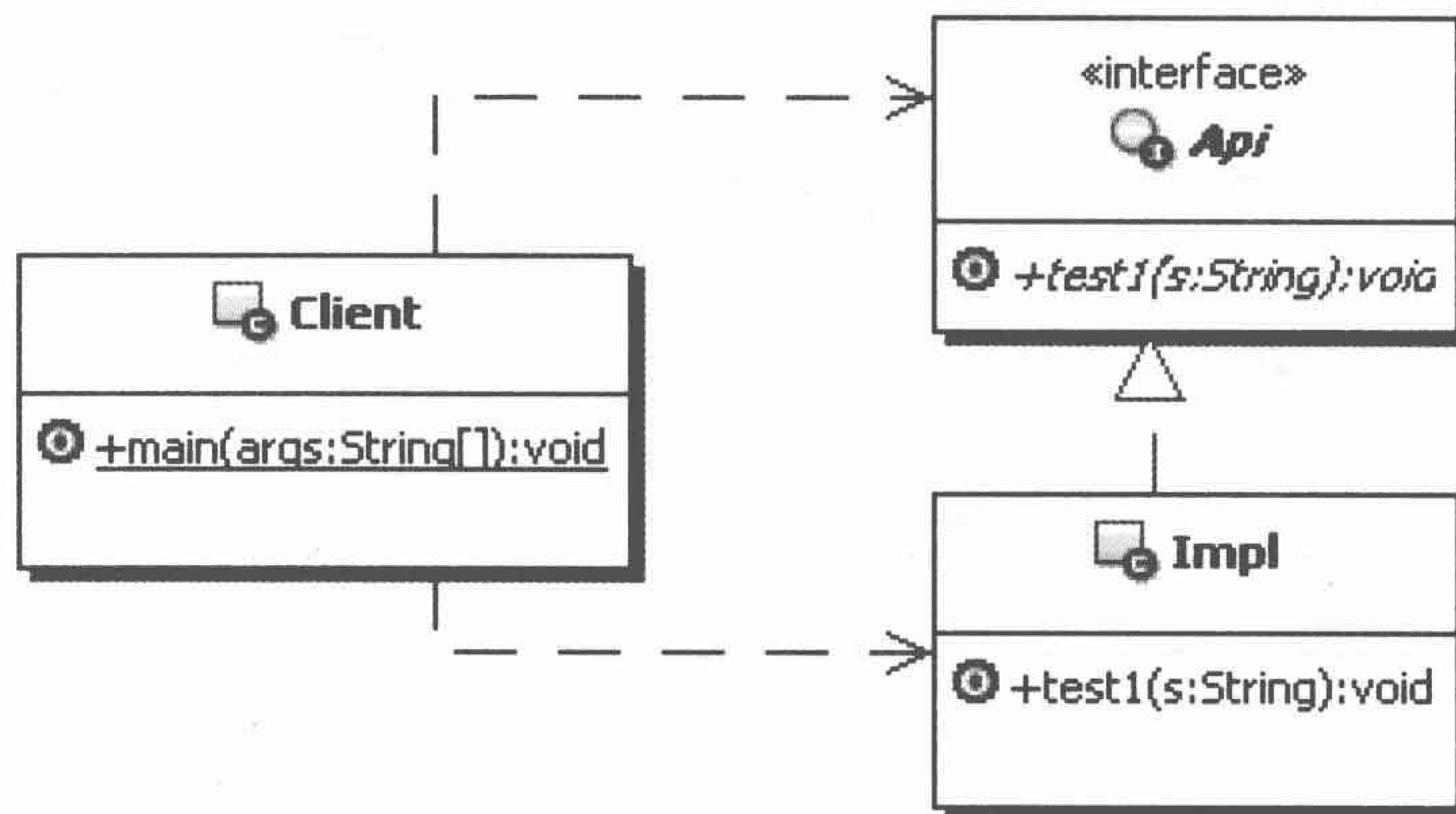


图 2.4　基本的接口和实现

用代码来说明会更清楚一些。

（1）首先定义接口 Api，示例代码如下：

```
/**
* 某个接口(通用的、抽象的、非具体的功能)
*/
public interface Api {
    /**
     * 某个具体的功能方法的定义，用 test1 来演示一下。
     * 这里的功能很简单，把传入的 s 打印输出即可
     * @param s 任意想要打印输出的字符串
     */
    public void test1(String s);
}
```

（2）既然有了接口，自然就要有实现，定义实现 Impl，示例代码如下：

```
/**
* 对接口的实现
*/
public class Impl implements Api{
    public void test1(String s) {
        System.out.println("Now In Impl. The input s=="+s);
    }
}
```

（3）那么此时的客户端怎么写呢？

按照 Java 的知识，接口不能直接使用，需要使用接口的实现类，示例代码如下：

```
/**
* 客户端：测试使用 Api 接口
*/
public class Client {
public static void main(String[] args) {
    Api api = new Impl();
    api.test1("哈哈，不要紧张，只是个测试而已！");
}
}
```

2.1.4　有何问题

上面写得没错吧，在 Java 的基础知识里面就是这么学的，难道这有什么问题吗？请仔细看位于客户端的下面这句话：

```
Api api = new Impl();
```

> **提示** **然后再想想接口的功能和思想，发现什么了？仔细再想想？**

你会发现在客户端调用的时候，客户端不但知道了接口，同时还知道了具体的实现就是 Impl。接口的思想是“封装隔离”，而实现类 Impl 应该是被接口 Api 封装并同客户端隔离开的，也就是说，客户端根本就不应该知道具体的实现类是 Impl。

有朋友说，那好，我就把 Impl 从客户端拿掉，让 Api 真正地对实现进行“封装隔离”，然后我们继续面向接口来编程。可是，新的问题出现了，当他把“new Impl()”去掉后，却发现无法得到 Api 接口对象了，怎么办呢？

把这个问题描述一下：在 Java 编程中，出现只知接口而不知实现，该怎么办？

就像现在的 Client，它知道要使用 Api 接口，但是不知由谁实现，也不知道如何实现，从而得不到接口对象，就无法使用接口，该怎么办呢？

2.2 解决方案

2.2.1 使用简单工厂来解决问题

用来解决上述问题的一个合理的解决方案就是简单工厂，那么什么是简单工厂呢？

1. 简单工厂的定义

> **提供一个创建对象实例的功能，而无须关心其具体实现。被创建实例的类型可以是接口、抽象类，也可以是具体的类。**

2. 应用简单工厂来解决问题的思路

分析上面的问题，虽然不能让模块外部知道模块内部的具体实现，但是模块内部是可以知道实现类的，而且创建接口是需要具体实现类的。

那么，干脆在模块内部新建一个类，在这个类里面来创建接口，然后把创建好的接口返回给客户端，这样，外部应用就只需要根据这个类来获取相应的接口对象，然后就可以操作接口定义的方法了。把这样的对象称为简单工厂，就叫它 Factory 吧。

这样一来，客户端就可以通过 Factory 来获取需要的接口对象，然后调用接口的方法来实现需要的功能，而且客户端也不用再关心具体的实现了。

2.2.2 简单工厂的结构和说明

简单工厂的结构如图 2.5 所示。

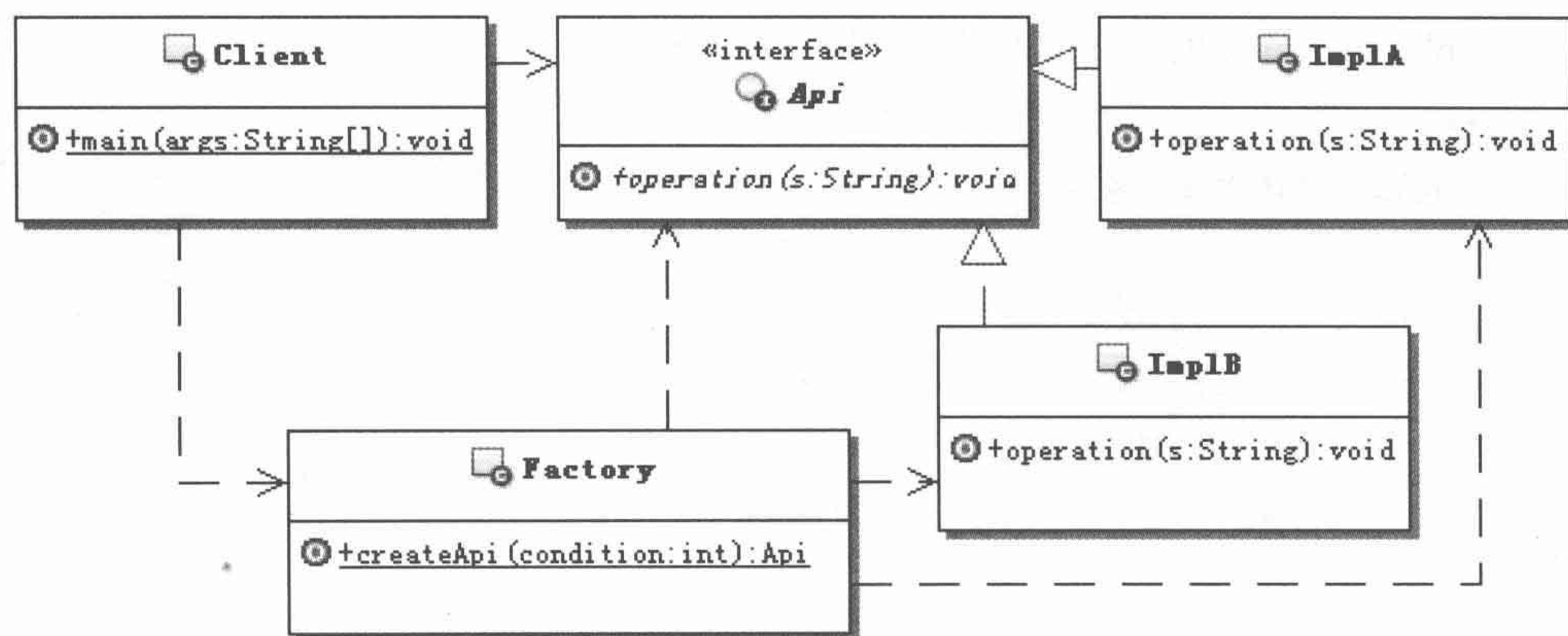


图 2.5　简单工厂的结构示意图

- Api：　定义客户所需要的功能接口。
- Impl：具体实现 Api 的实现类，可能会有多个。
- Factory：工厂，选择合适的实现类来创建 Api 接口对象。
- Client：客户端，通过 Factory 来获取 Api 接口对象，然后面向 Api 接口编程。

2.2.3　简单工厂示例代码

（1）Api 定义的示例代码如下：

```
/**
 * 接口的定义，该接口可以通过简单工厂来创建
 */
public interface Api {
    /**
     * 示意，具体功能方法的定义
     * @param s 示意，需要的参数
     */
    public void operation(String s);
}
```

（2）定义了接口，接下来实现它。ImplA 的示例代码如下：

```
/**
 * 接口的具体实现对象 A
 */
public class ImplA implements Api{
    public void operation(String s) {
        //实现功能的代码，示意一下
        System.out.println("ImplA s=="+s);
    }
}
```

ImplB 的示意实现和 ImplA 基本一样。示例代码如下：

```
/**
 * 接口的具体实现对象 B
 */
public class ImplB implements Api{
    public void operation(String s) {
        //实现功能的代码，示意一下
        System.out.println("ImplB s=="+s);
    }
}
```

（3）下面来看看简单工厂的实现。示例代码如下：

```
/**
 * 工厂类，用来创建 Api 对象
 */
public class Factory {
    /**
     * 具体创建 Api 对象的方法
     * @param condition 示意，从外部传入的选择条件
     * @return 创建好的 Api 对象
     */
    public static Api createApi(int condition){
        //应该根据某些条件去选择究竟创建哪一个具体的实现对象，
        //这些条件可以从外部传入，也可以从其他途径来获取。
        //如果只有一个实现，可以省略条件，因为没有选择的必要。
        //示意使用条件
        Api api = null;
        if(condition == 1){
            api = new ImplA();
        }else if(condition == 2){
            api = new ImplB();
        }
        return api;
    }
}
```

（4）再来看看客户端的示意，示例代码如下：

```
/**
 * 客户端，使用 Api 接口
 */
public class Client {
```

```
    public static void main(String[] args) {
        //通过简单工厂来获取接口对象
        Api api = Factory.createApi(1);
        api.operation("正在使用简单工厂");
    }
}
```

2.2.4　使用简单工厂重写示例

要使用简单工厂来重写前面的示例，主要就是要创建一个简单工厂对象，让简单工厂来负责创建接口对象。然后让客户端通过工厂来获取接口对象，而不再由客户端自己去创建接口的对象了。

此时系统的结构如图 2.6 所示。

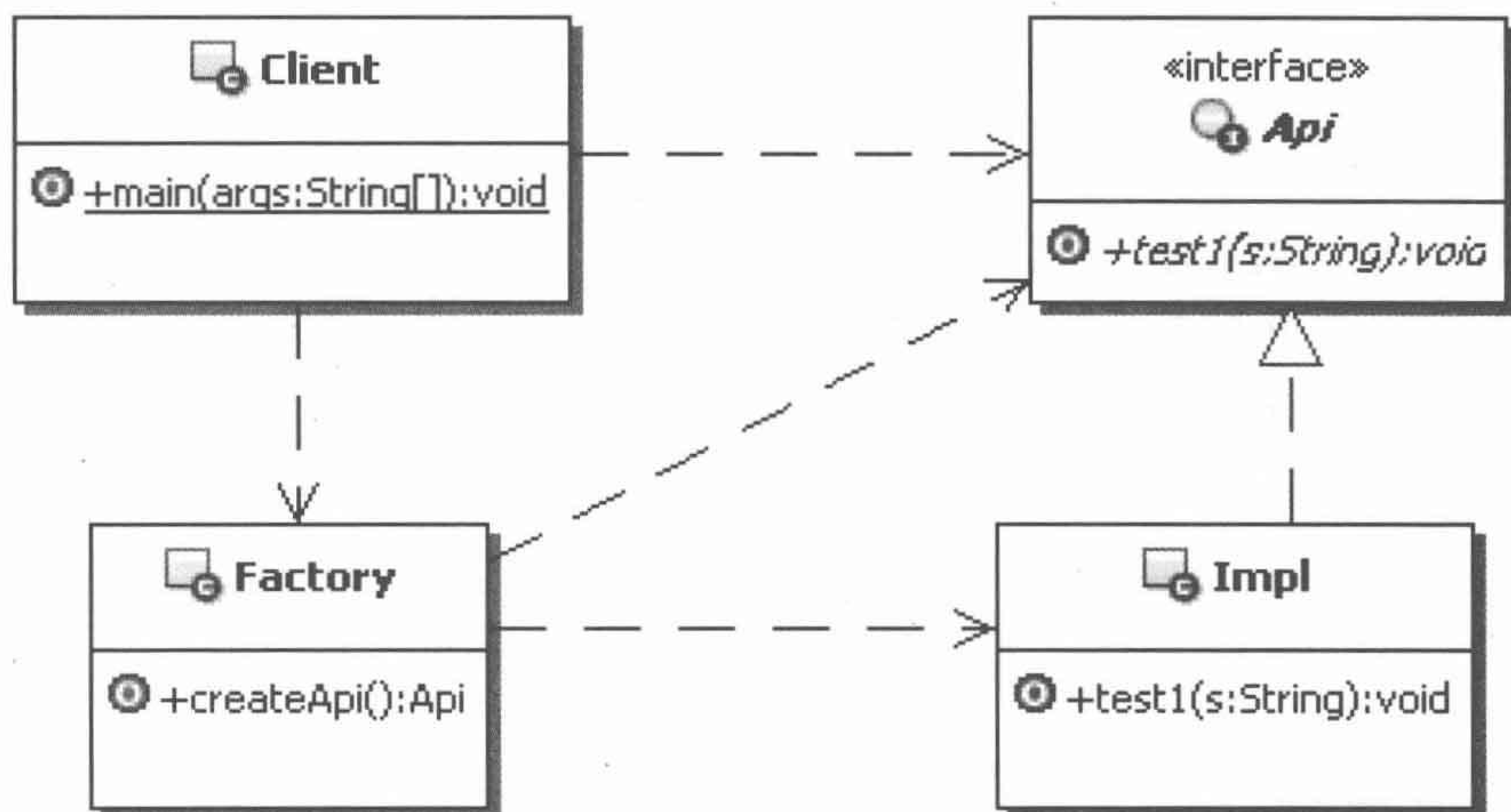


图 2.6　使用简单工厂重写示例的结构示意图

（1）接口 Api 和实现类 Impl 都和前面的示例一样，这里不再赘述。

（2）新创建一个简单工厂的对象。示例代码如下：

```
/**
 * 工厂类，用来创建 Api 对象
 */
public class Factory {
    /**
     * 具体创建 Api 对象的方法
     * @return 创建好的 Api 对象
     */
    public static Api createApi(){
        //由于只有一个实现，就不用条件来判断了
        return new Impl();
```

```
    }
}
```

（3）使用简单工厂。

客户端如何使用简单工厂提供的功能呢？这个时候，客户端就不用再自己去创建接口的对象了，应该使用工厂来获取。经过改造，客户端代码如下：

```
/**
* 客户端：测试使用 Api 接口
*/
public class Client {
    public static void main(String[] args) {
        //重要改变，没有new Impl()了，取而代之Factory.createApi()
      Api api = Factory.createApi();
        api.test1("哈哈，不要紧张，只是个测试而已！");
    }
}
```

就如同上面的示例，客户端通过简单工厂创建了一个实现接口的对象，然后面向接口编程，从客户端来看，它根本不知道具体的实现是什么，也不知道是如何实现的，它只知道通过工厂获得了一个接口对象，然后通过这个接口来获取想要的功能。

事实上，简单工厂能帮助我们真正地开始面向接口编程，像以前的做法，其实只是用到了接口的多态部分的功能，而最重要的“封装隔离性”并没有体现出来。

2.3 模式讲解

2.3.1 典型疑问

首先来解决一个常见的问题：可能有朋友会认为，上面示例中的简单工厂看起来不就是把客户端里面的“new Impl()”移动到简单工厂里面吗？不还是一样通过 new 一个实现类来得到接口吗？把“new Impl()”这句话放到客户端和放到简单工厂里面有什么不同吗？

提示	**理解这个问题的重点就在于理解简单工厂所处的位置。**

根据前面的学习，我们知道接口是用来封装隔离具体的实现的，目标就是不要让客户端知道封装体内部的具体实现。简单工厂的位置是位于封装体内的，也就是简单工厂是跟接口和具体的实现在一起的，算是封装体内部的一个类，所以简单工厂知道具体的实现类是没有关系的。重新整理一下简单工厂的结构图，如图 2.7 所示。

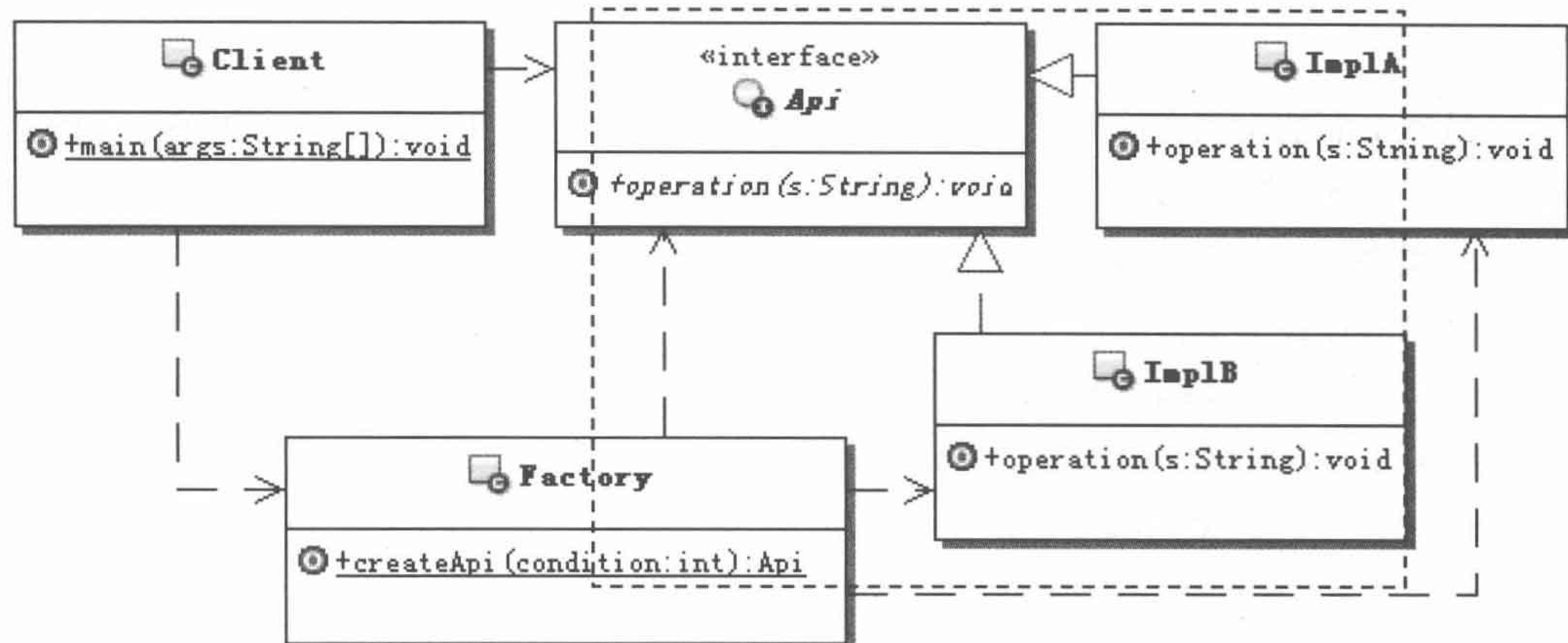


图 2.7　整理后的简单工厂结构

图 2.7 中的虚线框，就好比是一个组件的包装边界，表示接口、实现类和工厂类组合成了一个组件。在这个封装体里面，只有接口和工厂是对外的，也就是让外部知道并使用的，所以故意漏了一些在虚线框外，而具体的实现类是不对外的，被完全包含在虚线框内。

对于客户端而言，只是知道了接口 Api 和简单工厂 Factory，通过 Factory 就可以获得 Api 了，这样就达到了让 Client 在不知道具体实现类的情况下获取接口 Api。

所以看似简单地将 new Impl()这句话从客户端里面移动到了简单工厂里面，其实是有了质的变化的。

2.3.2　认识简单工厂

1. 简单工厂的功能

工厂嘛，就是用来创造东西的。在 Java 里面，通常情况下是用来创造接口的，但是也可以创造抽象类，甚至是一个具体的类实例。

> 注意　**一定要注意，虽然前面的示例是利用简单工厂来创建的接口，但是也可以用简单工厂来创建抽象类或普通类的实例。**

2. 静态工厂

使用简单工厂的时候，通常不用创建简单工厂类的类实例，没有创建实例的必要。因此可以把简单工厂类实现成一个工具类，直接使用静态方法就可以了。也就是说简单工厂的方法通常是静态的，所以也被称为静态工厂。如果要防止客户端无谓地创造简单工厂实例，还可以把简单工厂的构造方法私有化了。

3. 万能工厂

一个简单工厂可以包含很多用来构造东西的方法，这些方法可以创建不同的接口、抽象类或者是类实例。一个简单工厂理论上可以构造任何东西，所以又称之为“万能工

厂”。

虽然上面的实例在简单工厂里面只有一个方法，但事实上，是可以有很多这样的创建方法的，这点要注意。

4. 简单工厂创建对象的范围

虽然从理论上讲，简单工厂什么都能创建，但对于简单工厂可创建对象的范围，通常不要太大，建议控制在一个独立的组件级别或者一个模块级别，也就是一个组件或模块简单工厂。否则这个简单工厂类会职责不明，有点大杂烩的感觉。

5. 简单工厂的调用顺序示意图

简单工厂的调用顺序如图 2.8 所示：

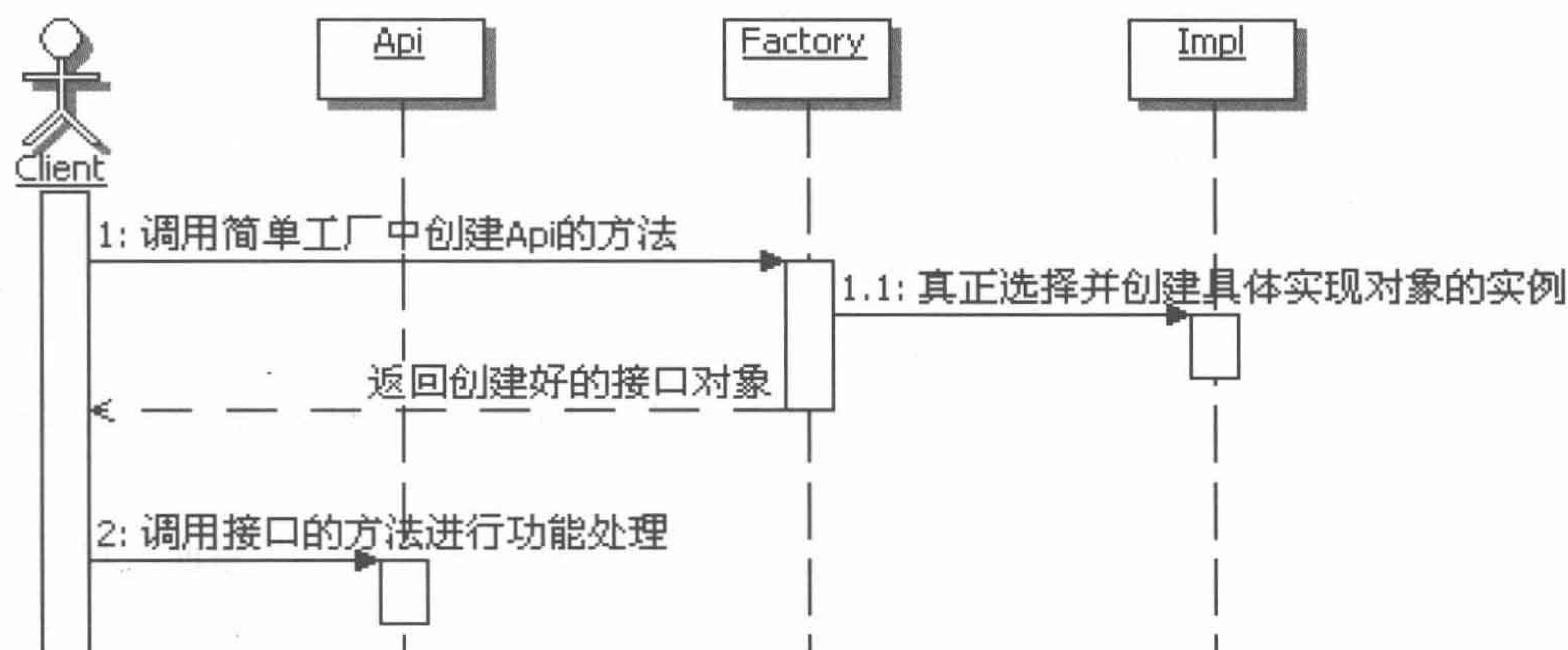


图 2.8　简单工厂的调用顺序示意图

6. 简单工厂命名的建议

- 类名称建议为“模块名称+Factory”。比如，用户模块的工厂就称为 UserFactory。
- 方法名称通常为“get+接口名称”或者是“create+接口名称”。比如，有一个接口名称为 UserEbi，那么方法名称通常为 getUserEbi 或者是 createUserEbi。

当然，也有一些朋友习惯于把方法名称命名为“new+接口名称”。比如，newUserEbi，我们不提倡这样做。因为 new 在 Java 中代表特定的含义，而且通过简单工厂的方法来获取对象实例，并不一定每次都是要 new 一个新的实例。如果使用 newUserEbi，会给人错觉，好像每次都是 new 一个新的实例一样。

2.3.3　简单工厂中方法的写法

虽然说简单工厂的方法大多是用来创建接口的，但是仔细分析就会发现，真正能实现功能的是具体的实现类，这些实现类是已经做好的，并不是真的要靠简单工厂来创造出来的，简单工厂的方法无外乎就是：实现了选择一个合适的实现类来使用。

所以说简单工厂方法的内部主要实现的功能是**“选择合适的实现类”**来创建实例对象。既然要实现选择，那么就需要选择的条件或者是选择的参数，选择条件或者是参数的来源通常又有以下几种。

- 来源于客户端，由 Client 来传入参数
- 来源于配置文件，从配置文件获取用于判断的值
- 来源于程序运行期的某个值，比如从缓存中获取某个运行期的值

下面来看示例，看看由客户端来传入参数，如何写简单工厂中的方法。

（1）在 2.3.3 节的示例上再添加一个实现，称为 Impl2，示例代码如下：

```
/**
 * 对接口的一种实现
 */
public class Impl2 implements Api{
    public void test1(String s) {
        System.out.println("Now In Impl2. The input s=="+s);
    }
}
```

（2）现在对 Api 这个接口，有了两种实现，那么工厂类该怎么办呢？到底如何选择呢？不可能两个同时使用吧，看看新的工厂类，示例代码如下：

```
/**
 * 工厂类，用来创建 Api 的
 */
public class Factory {
    /**
     * 具体创建 Api 的方法，根据客户端的参数来创建接口
     * @param type 客户端传入的选择创建接口的条件
     * @return 创建好的 Api 对象
     */
    public static Api createApi(int type){
        //这里的 type 也可以不由外部传入，而是直接读取配置文件来获取
        //为了把注意力放在模式本身上，这里就不去写读取配置文件的代码了
        //根据 type 来进行选择，当然这里的 1 和 2 应该作为常量
        Api api = null;
        if(type==1){
            api = new Impl();
        }else if(type==2){
            api = new Impl2();
        }
        return api;
    }
}
```

注意这里添加了参数

选择究竟使用哪一个具体的实现

（3）客户端没有什么变化，只是在调用 Factory 的 createApi 方法的时候需要传入参数，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //注意这里传递的参数，修改参数就可以修改行为，试试看吧
        Api api = Factory.createApi(2);
        api.test1("哈哈，不要紧张，只是个测试而已！");
    }
}
```

（4）要注意这种方法有一个缺点。

由于是从客户端在调用工厂的时候传入选择的参数，这就说明客户端必须知道每个参数的含义，也需要理解每个参数对应的功能处理。这就要求必须在一定程度上，向客户暴露一定的内部实现细节。

2.3.4 可配置的简单工厂

现在已经学会通过简单工厂来选择具体的实现类了，可是还有问题。比如，在现在的实现中，再新增加一种实现，该怎么办呢？

那就需要修改工厂类，才能把新的实现添加到现有的系统中。比如现在新增加了一个实现类 Impl3，那么就需要类似下面这样来修改工厂类：

```
public class Factory {
    public static Api createApi(int type){
        Api api = null;
        if(type==1){
            api = new Impl();
        }else if(type==2){
            api = new Impl2();
        }
        else if(type==3){
            api = new Impl3();
        }
        return api;
    }
}
```


每次新增加一个实现类都来修改工厂类的实现，肯定不是一个好的实现方式。那么现在希望新增加了实现类过后不修改工厂类，该怎么办呢？

一个解决的方法就是使用配置文件，当有了新的实现类后，只要在配置文件里面配置上新的实现类即可。在简单工厂的方法里面可以使用反射，当然也可以使用 IoC/DI（控制反转/依赖注入，这个不在这里讨论）来实现。

下面来看看如何使用反射加上配置文件，来实现添加新的实现类后，无须修改代码，就能把这个新的实现类加入应用中。

（1）配置文件用最简单的 properties 文件，实际开发中多是 xml 配置。定义一个名称为“FactoryTest.properties”的配置文件，放置到 Factory 同一个包下面，内容如下：

```
ImplClass=cn.javass.dp.simplefactory.example5.Impl
```

如果新添加了实现类，修改这里的配置即可，就不需要修改程序了。

（2）此时的工厂类实现如下：

```
/**
* 工厂类，用来创建 Api 对象
*/
public class Factory {
    /**
     * 具体创建 Api 的方法，根据配置文件的参数来创建接口
     * @return 创建好的 Api 对象
     */
    public static Api createApi(){
        //直接读取配置文件来获取需要创建实例的类
        //至于如何读取 Properties，还有如何反射在这里就不解释了
        Properties p = new Properties();
        InputStream in = null;
        try {
            in = Factory.class.getResourceAsStream(
                                    "FactoryTest.properties");
            p.load(in);
        } catch (IOException e) {
            System.out.println(
                        "装载工厂配置文件出错了，具体的堆栈信息如下：");
            e.printStackTrace();
        }finally{
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        //用反射去创建，那些例外处理等完善的工作这里就不做了
        Api api = null;
        try {
            api = (Api)Class.forName(p.getProperty("ImplClass"))
                                              .newInstance();
        } catch (InstantiationException e) {
```

```
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return api;
    }
}
```

（3）此时的客户端就变得很简单了，不再需要传入参数，代码示例如下：

```
public class Client {
    public static void main(String[] args) {
        Api api = Factory.createApi();
        api.test1("哈哈，不要紧张，只是个测试而已！");
    }
}
```


把上面的示例代码输入到电脑里面，测试一下，体会体会。

2.3.5 简单工厂的优缺点

简单工厂有以下优点。

- 帮助封装

  简单工厂虽然很简单，但是非常友好地帮助我们实现了组件的封装，然后让组件外部能真正面向接口编程。

- 解耦

  通过简单工厂，实现了客户端和具体实现类的解耦。

  如同上面的例子，客户端根本就不知道具体是由谁来实现，也不知道具体是如何实现的，客户端只是通过工厂获取它需要的接口对象。

简单工厂有以下缺点。

- 可能增加客户端的复杂度

  如果通过客户端的参数来选择具体的实现类，那么就必须让客户端能理解各个参数所代表的具体功能和含义，这样会增加客户端使用的难度，也部分暴露了内部实现，这种情况可以选用可配置的方式来实现。

- 不方便扩展子工厂

  私有化简单工厂的构造方法，使用静态方法来创建接口，也就不能通过写简单工厂类的子类来改变创建接口的方法的行为了。不过，通常情况下是不需要为简单工厂创建子类的。

2.3.6　思考简单工厂

1. 简单工厂的本质

> **简单工厂的本质是：选择实现**

注意简单工厂的重点在选择，实现是已经做好了的。就算实现再简单，也要由具体的实现类来实现，而不是在简单工厂里面来实现。简单工厂的目的在于为客户端来选择相应的实现，从而使得客户端和实现之间解耦。这样一来，具体实现发生了变化，就不用变动客户端了，这个变化会被简单工厂吸收和屏蔽掉。

实现简单工厂的难点就在于 “如何选择”实现，前面讲到了几种传递参数的方法，那都是静态的参数，还可以实现成为动态的参数。比如，在运行期间，由工厂去读取某个内存的值，或者是去读取数据库中的值，然后根据这个值来选择具体的实现等。

2. 何时选用简单工厂

建议在以下情况中选用简单工厂。

- 如果想要完全封装隔离具体实现，让外部只能通过接口来操作封装体，那么可以选用简单工厂，让客户端通过工厂来获取相应的接口，而无须关心具体的实现。
- 如果想要把对外创建对象的职责集中管理和控制，可以选用简单工厂，一个简单工厂可以创建很多的、不相关的对象，可以把对外创建对象的职责集中到一个简单工厂来，从而实现集中管理和控制。

2.3.7　相关模式

- 简单工厂和抽象工厂模式
  简单工厂是用来选择实现的，可以选择任意接口的实现。一个简单工厂可以有多个用于选择并创建对象的方法，多个方法创建的对象可以有关系也可以没有关系。
  抽象工厂模式是用来选择产品簇的实现的，也就是说一般抽象工厂里面有多个用于选择并创建对象的方法，但是这些方法所创建的对象之间通常是有关系的，这些被创建的对象通常是构成一个产品簇所需要的部件对象。
  所以从某种意义上来说，简单工厂和抽象工厂是类似的，如果抽象工厂退化成为只有一个实现，不分层次，那么就相当于简单工厂了。
- 简单工厂和工厂方法模式
  简单工厂和工厂方法模式也是非常类似的。
  工厂方法的本质也是用来选择实现的，跟简单工厂的区别在于工厂方法是把选择具体实现的功能延迟到子类去实现。
  如果把工厂方法中选择的实现放到父类直接实现，那就等同于简单工厂。

- 简单工厂和能创建对象实例的模式

  简单工厂的本质是选择实现，所以它可以跟其他任何能够具体的创建对象实例的模式配合使用，比如：单例模式、原型模式、生成器模式等。

第3章　外观模式（Facade）

3.1 场景问题

3.1.1 生活中的示例

外观模式在现实生活中的示例很多，比如组装电脑，通常会有两种方案。

一个方案是去电子市场把自己需要的配件都买回来，然后自己组装，绝对 DIY（Do It Yourself）。这个方案好是好，但是需要对各种配件都要比较熟悉，这样才能选择最合适的配件，而且还要考虑配件之间的兼容性，如图 3.1 所示。

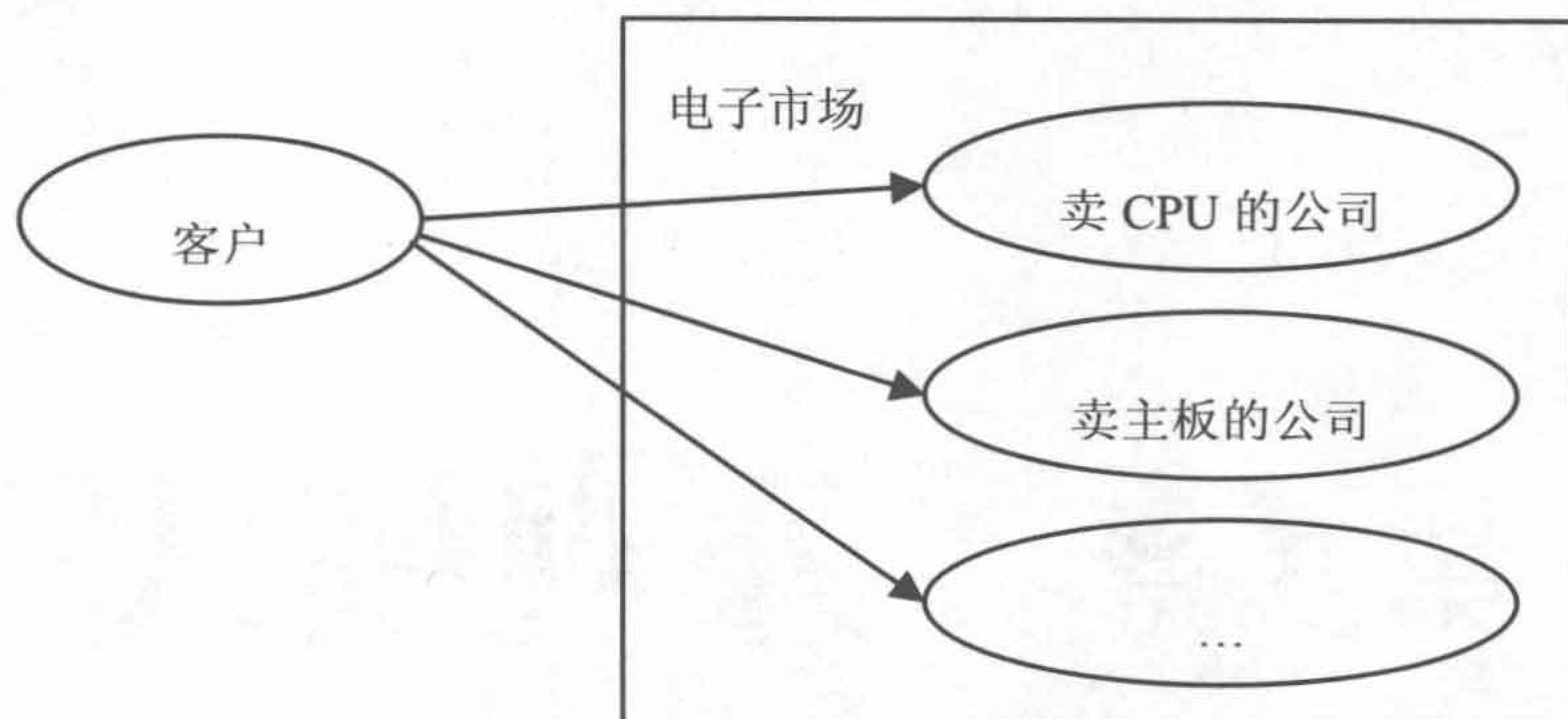


图 3.1　客户完全自己组装电脑

另外一个方案，就是到电子市场，找一家专业的装机公司，把具体的要求提出来，然后等着拿电脑就好了。当然价格会比自己全部 DIY 贵一些，但综合起来还算是个不错的选择，这也是大多数人的选择，如图 3.2 所示。

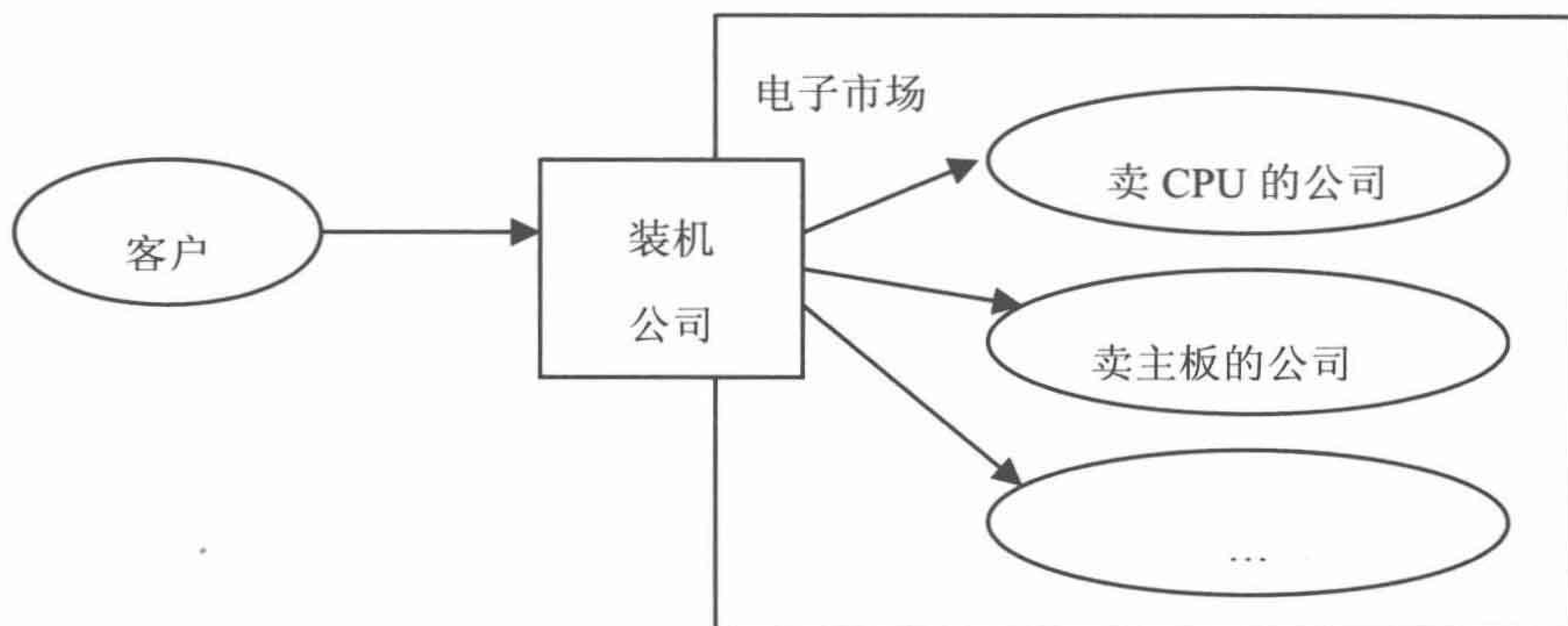


图 3.2　找专业的装机公司组装电脑

这个专业的装机公司就相当于本章的主角——外观模式（Facade）。有了它，我们就不用自己去和众多卖配件的公司打交道，只需要跟装机公司交互就可以了，并将组装好的电脑返回给我们。

把上面的过程抽象一下，如果把电子市场看成是一个系统，而各个卖配件的公司看成是模块的话，就类似于出现了这样一种情况：客户端为了完成某个功能，需要去调用某个系统中的多个模块，把它们称为 A 模块、B 模块和 C 模块。对于客户端而言，那就需要知道 A、B、C 这三个模块的功能，还需要知道如何组合这多个模块提供的功能来实

现自己所需要的功能，非常麻烦。

要是有一个简单的方式能让客户端去实现相同的功能该多好啊，这样，客户端就不用跟系统中的多个模块交互，而且客户端也不需要知道那么多模块的细节功能了，实现这个功能的就是 Facade。

3.1.2　代码生成的应用

考虑这样一个实际的应用：代码生成。

很多公司都有这样的应用工具，能根据配置生成代码。一般这种工具都是公司内部使用，较为专有的工具，生成的多是按照公司的开发结构来实现的常见的基础功能，比如增删改查。这样每次在开发实际应用的时候，就可以以很快的速度把基本的增删改查实现出来，然后把主要的精力都放在业务功能的实现上。

当然这里不可能去实现一个这样的代码生成工具，那需要整本书的内容，这里仅用它来说明外观模式。

假设使用代码生成出来的每个模块都具有基本的三层架构，分为表现层、逻辑层和数据层，那么代码生成工具里面就应该有相应的代码生成处理模块。

另外，代码生成工具自身还需要一个配置管理的模块，通过配置来告诉代码生成工具，每个模块究竟需要生成哪些层的代码。比如，通过配置来描述，是只需要生成表现层代码呢，还是三层都生成。具体的模块示意如图 3.3 所示。

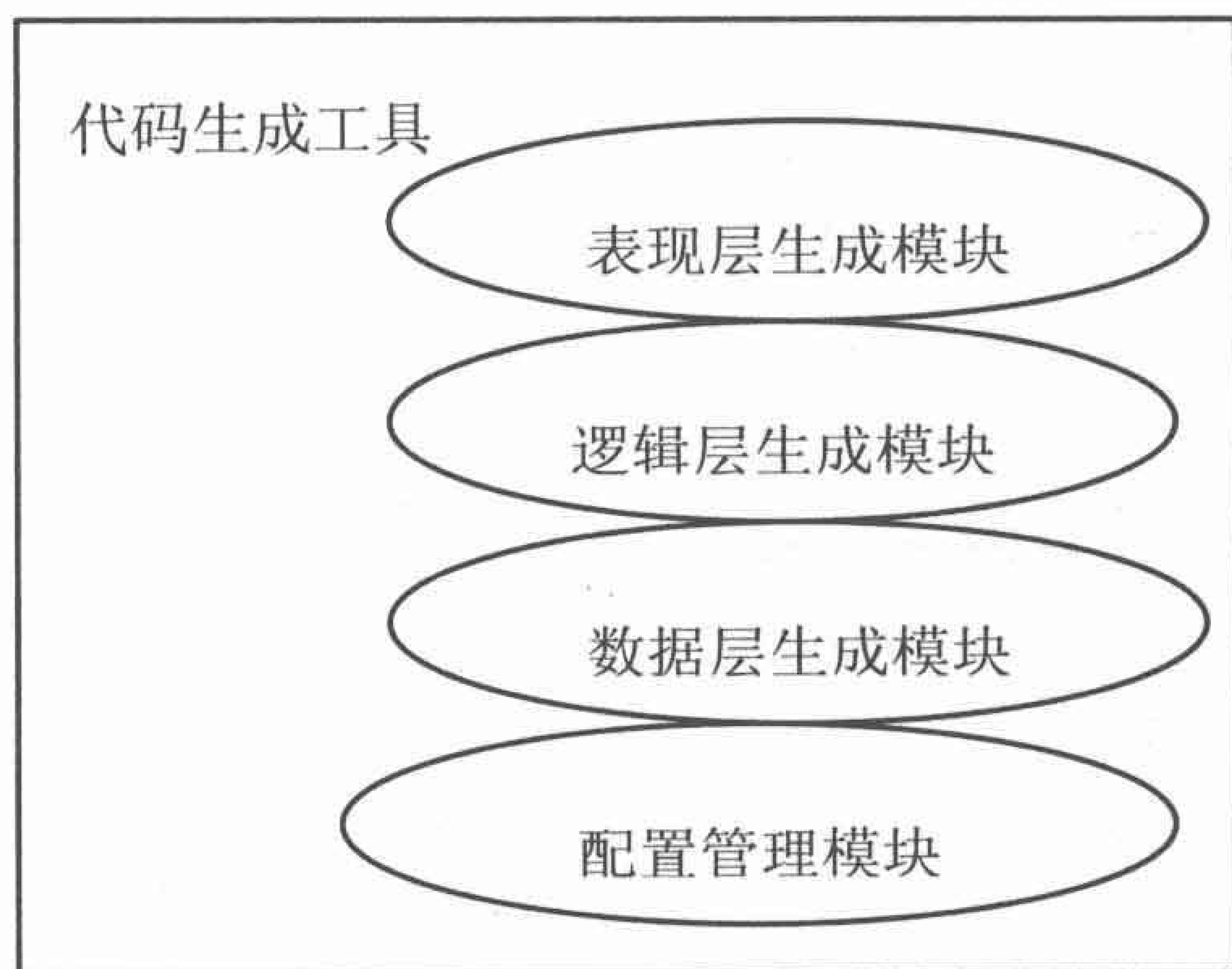


图 3.3　代码生成工具的模块示意图

那么现在客户端需要使用这个代码生成工具来生成需要的基础代码，该如何实现呢？

3.1.3　不用模式的解决方案

有朋友会想，开发一个这样的工具或许会比较麻烦，但是使用一下，应该不难吧，直接调用不就可以了。

在示范客户端之前，先来把工具模拟示范出来。为了简单，每个模块就写一个类，

而且每个类只是实现一个功能，仅仅做一个示范。

（1）先看看描述配置的数据 Model。示例代码如下：

```
/**
 * 示意配置描述的数据 Model，真实的配置数据会很多
 */
public class ConfigModel {
    /**
     * 是否需要生成表现层，默认是 true
     */
    private boolean needGenPresentation = true;
    /**
     * 是否需要生成逻辑层，默认是 true
     */
    private boolean needGenBusiness = true;
    /**
     * 是否需要生成 DAO，默认是 true
     */
    private boolean needGenDAO = true;
    public boolean isNeedGenPresentation() {
        return needGenPresentation;
    }
    public void setNeedGenPresentation(
                                boolean needGenPresentation) {
        this.needGenPresentation = needGenPresentation;
    }
    public boolean isNeedGenBusiness() {
        return needGenBusiness;
    }
    public void setNeedGenBusiness(boolean needGenBusiness) {
        this.needGenBusiness = needGenBusiness;
    }
    public boolean isNeedGenDAO() {
        return needGenDAO;
    }
    public void setNeedGenDAO(boolean needGenDAO) {
        this.needGenDAO = needGenDAO;
    }
}
```

（2）接下来看看配置管理的实现示意。示例代码如下：

```
/**
 * 示意配置管理，就是负责读取配置文件，
 * 并把配置文件的内容设置到配置 Model 中去，是个单例
 */
public class ConfigManager {
    private static ConfigManager manager = null;
    private static ConfigModel cm = null;
    private ConfigManager(){
        //
    }
    public static ConfigManager getInstance(){
        if(manager == null){
            manager = new ConfigManager();
            cm = new ConfigModel();
            //读取配置文件，把值设置到 ConfigModel 中去，这里省略了
        }
        return manager;
    }
    /**
     * 获取配置的数据
     * @return 配置的数据
     */
    public ConfigModel getConfigData(){
        return cm;
    }
}
```

（3）再来看看各个生成代码的模块。在示意中，它们的实现类似，就是获取配置文件的内容，然后按照配置来生成相应的代码。

先来看生成表现层的示意实现。示例代码如下：

```
/**
 * 示意生成表现层的模块
 */
public class Presentation {
    public void generate(){
        //1：从配置管理里面获取相应的配置信息
        ConfigModel cm =
                    ConfigManager.getInstance().getConfigData();
        if(cm.isNeedGenPresentation()){
            //2：按照要求去生成相应的代码，并保存成文件
```

```
            System.out.println("正在生成表现层代码文件");
        }
    }
}
```

再来看生成逻辑层的示意实现。示例代码如下：

```
/**
 * 示意生成逻辑层的模块
 */
public class Business {
    public void generate(){
        ConfigModel cm =
                    ConfigManager.getInstance().getConfigData();
        if(cm.isNeedGenBusiness()){
            System.out.println("正在生成逻辑层代码文件");
        }
    }
}
```

下面是生成数据层的示意实现。示例代码如下：

```
/**
 * 示意生成数据层的模块
 */
public class DAO {
    public void generate(){
        ConfigModel cm =
                    ConfigManager.getInstance().getConfigData();
        if(cm.isNeedGenDAO()){
            System.out.println("正在生成数据层代码文件");
        }
    }
}
```

（4）此时的客户端实现，就应该自行去调用这多个模块了。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //现在没有配置文件，直接使用默认的配置，通常情况下，三层都应该生成
        //也就是说客户端必须对这些模块都有了解，才能够正确使用它们
        new Presentation().generate();
        new Business().generate();
        new DAO().generate();
    }
```

```
}
```

运行结果如下：

```
正在生成表现层代码文件
正在生成逻辑层代码文件
正在生成数据层代码文件
```

3.1.4 有何问题

仔细查看上面的实现，会发现其中有一个问题：那就是客户端为了使用生成代码的功能，需要与生成代码子系统内部的多个模块交互。

这对于客户端而言，是个麻烦，使得客户端不能简单地使用生成代码的功能。而且，如果其中的某个模块发生了变化，还可能会引起客户端也要随着变化。

那么如何实现，才能让子系统外部的客户端在使用子系统的时候，既能简单地使用这些子系统内部的模块功能，而又不用客户端去与子系统内部的多个模块交互呢？

3.2 解 决 方 案

3.2.1 使用外观模式来解决问题

用来解决上述问题的一个合理的解决方案就是外观模式。那么什么是外观模式呢？

1. 外观模式的定义

这里先对两个词进行一下说明，一个是界面，另一个是接口。

> **为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。**

1）界面

一提到界面，估计很多朋友的第一反应就是图形界面（GUI）。其实在这里提到的界面，主要指的是从一个组件外部来看这个组件，能够看到什么，这就是这个组件的界面，也就是所说的外观。

比如，你从一个类外部来看这个类，那么这个类的 public 方法就是这个类的外观，因为你从类外部来看这个类，就能看到这些。

再比如，你从一个模块外部来看这个模块，那么这个模块对外的接口就是这个模块的外观，因为你只能看到这些接口，其他的模块内部实现的部分是被接口封装隔离了的。

2）接口

一提到接口，做 Java 的朋友的第一反应就是 interface。其实在这里提到的接口，主

要指的是外部和内部交互的这么一个通道，通常是指一些方法，可以是类的方法，也可以是 interface 的方法。也就是说，这里所说的接口，并不等价于 interface，也有可能是一个类。

2. 应用外观模式来解决问题的思路

仔细分析上面的问题，客户端想要操作更简单点，那就根据客户端的需要来给客户端定义一个简单的接口，然后让客户端调用这个接口，剩下的事情客户端就不用管它，这样客户端就变得简单了。

当然，这里所说的接口就是客户端和被访问的系统之间的一个通道，并不一定是指 Java 的 interface。它在外观模式里面，通常指的是类，这个类被称为“外观”。

外观模式就是通过引入这么一个外观类，在这个类里面定义客户端想要的简单的方法，然后在这些方法的实现里面，由外观类再去分别调用内部的多个模块来实现功能，从而让客户端变得简单。这样一来，客户端就只需要和外观类交互就可以了。

3.2.2 外观模式的结构和说明

外观模式的结构如图 3.4 所示。

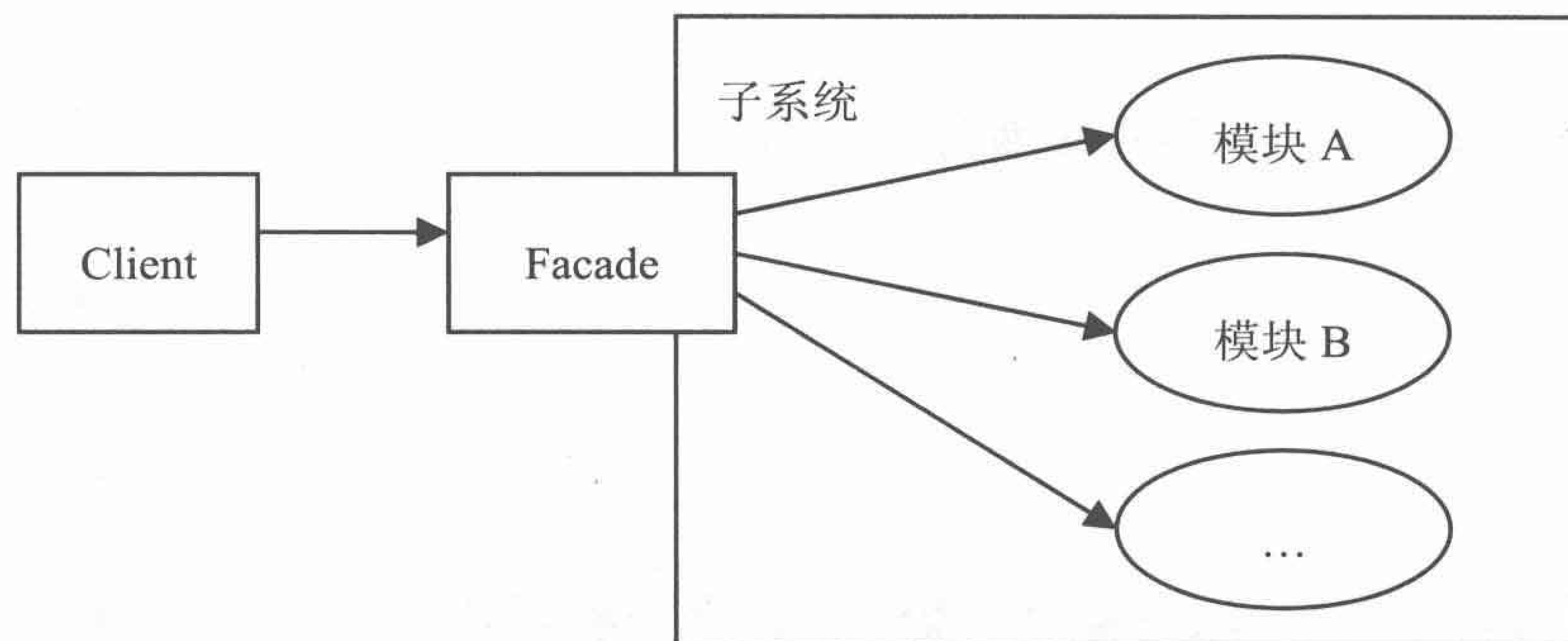


图 3.4 外观模式结构示意图

1. Facade

定义子系统的多个模块对外的高层接口，通常需要调用内部多个模块，从而把客户的请求代理给适当的子系统对象。

2. 模块

接受 Facade 对象的委派，真正实现功能，各个模块之间可能有交互。

但是请注意，Facade 对象知道各个模块，但是各个模块不应该知道 Facade 对象。

3.2.3 外观模式示例代码

由于外观模式的结构图过于抽象，因此把它稍稍具体点。假设子系统内有三个模块，分别是 AModule、BModule 和 CModule，它们分别有一个示意的方法，那么此时示例的整体结构如图 3.5 所示。

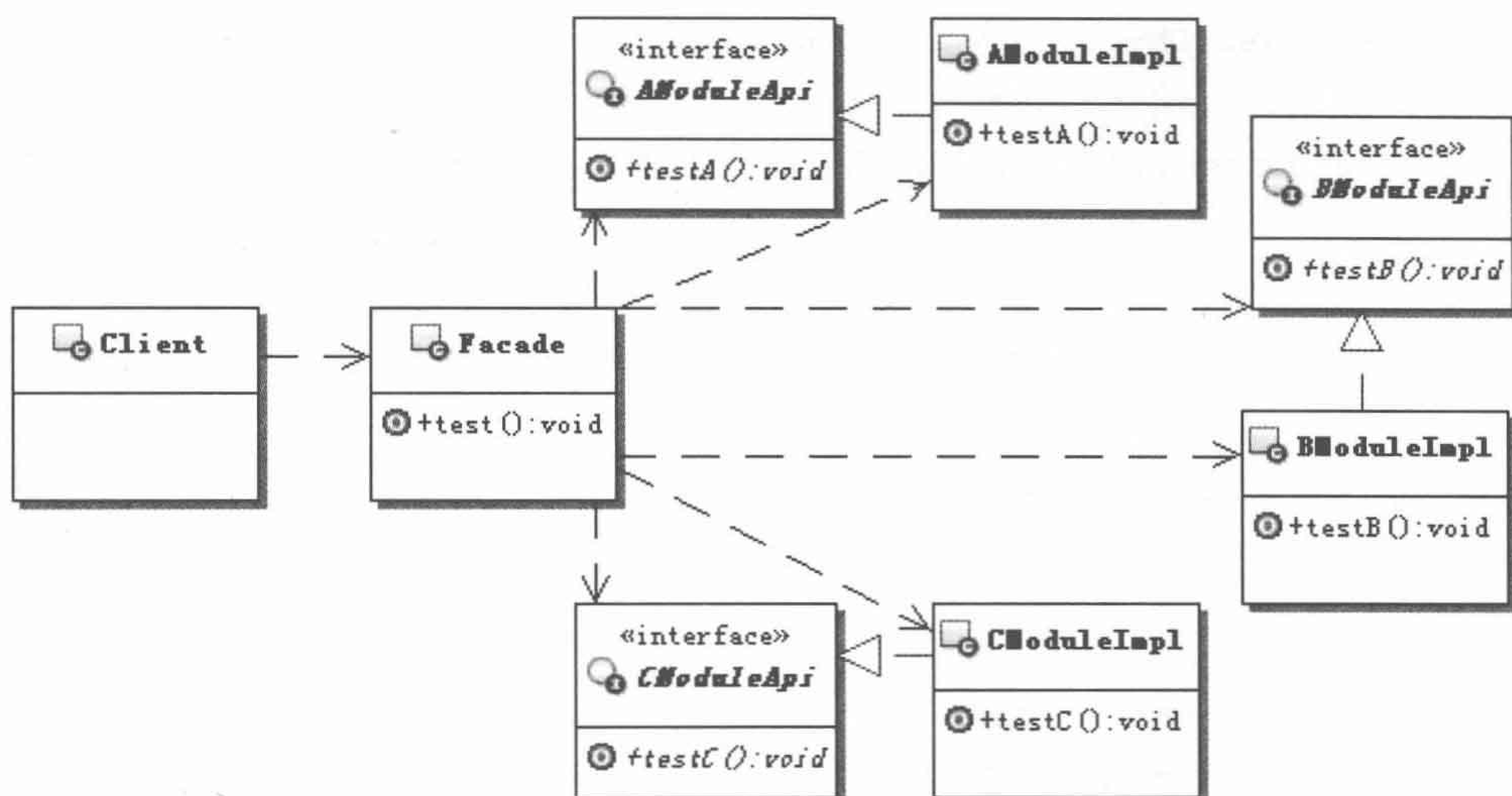


图 3.5　外观模式示例的整体结构示意图

还是来看看代码示例，会比较清楚。

（1）首先定义 A 模块的接口，A 模块对外提供功能方法，从抽象的高度去看，可以是任意的功能方法。示例代码如下：

```
/**
 * A 模块的接口
 */
public interface AModuleApi {
    /**
     * 示意方法，A 模块对外的一个功能方法
     */
    public void testA();
}
```

（2）实现 A 模块的接口。简单示范一下，示例代码如下：

```
public class AModuleImpl implements AModuleApi{
    public void testA() {
        System.out.println("现在在 A 模块里面操作 testA 方法");
    }
}
```

（3）同理，定义和实现 B 模块和 C 模块。

先来看 B 模块的接口定义。示例代码如下：

```
public interface BModuleApi {
    public void testB();
}
```

B 模块的实现示意，示例代码如下：

```
public class BModuleImpl implements BModuleApi{
    public void testB() {
        System.out.println("现在在B模块里面操作testB方法");
    }
}
```

C模块的接口定义，示例代码如下：

```
public interface CModuleApi {
    public void testC();
}
```

C模块的实现示意，示例代码如下：

```
public class CModuleImpl implements CModuleApi{
    public void testC() {
        System.out.println("现在在C模块里面操作testC方法");
    }
}
```

（4）定义外观对象，示例代码如下：

```
/**
 * 外观对象
 */
public class Facade {
    /**
     * 示意方法，满足客户需要的功能
     */
    public void test(){
        //在内部实现的时候，可能会调用到内部的多个模块
        AModuleApi a = new AModuleImpl();
        a.testA();
        BModuleApi b = new BModuleImpl();
        b.testB();
        CModuleApi c = new CModuleImpl();
        c.testC();
    }
}
```

（5）客户端如何使用呢？直接使用外观对象就可以了。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //使用Facade
        new Facade().test();
```

```
    }
}
```

运行结果如下：

```
现在在 A 模块里面操作 testA 方法
现在在 B 模块里面操作 testB 方法
现在在 C 模块里面操作 testC 方法
```

3.2.4　使用外观模式重写示例

要使用外观模式重写前面的示例，其实非常简单，只须添加一个 Facade 的对象，然后在里面实现客户端需要的功能就可以了。

（1）新添加一个 Facade 对象。示例代码如下：

```
/**
 * 代码生成子系统的外观对象
 */
public class Facade {
    /**
     * 客户端需要的，一个简单的调用代码生成的功能
     */
    public void generate(){
        new Presentation().generate();
        new Business().generate();
        new DAO().generate();
    }
}
```

（2）其他定义和实现都没有变化，这里就不再赘述。

（3）看看此时的客户端怎么实现？不再需要客户端去调用子系统内部的多个模块，直接使用外观对象就可以了。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
       //使用 Facade
        new Facade().generate();
    }
}
```

去运行看看，是否能正确地实现功能。

如同上面讲述的例子，Facade 类其实相当于 A、B、C 模块的外观界面，Facade 类也被称为 A、B、C 模块对外的接口，有了这个 Facade 类，那么客户端就不需要知道系统内部的实现细节，甚至客户端都不需要知道 A、B、C 模块的存在，客户端只需要跟

Facade 类交互就好了，从而更好地实现了客户端和子系统中 A、B、C 模块的解耦，让客户端更容易地使用系统。

3.3 模式讲解

3.3.1 认识外观模式

1. 外观模式的目的

外观模式的目的不是给子系统添加新的功能接口，而是为了让外部减少与子系统内多个模块的交互，松散耦合，从而让外部能够更简单地使用子系统。

这点要特别注意，因为外观是当作子系统对外的接口出现的，虽然也可以在这里定义一些子系统没有的功能，但不建议这么做。外观应该是包装已有的功能，它主要负责组合已有功能来实现客户需要，而不是添加新的实现。

2. 使用外观和不使用外观相比有何变化

> 提示　**看到 Facade 的实现，可能有些朋友会说，这不就是把原来在客户端的代码搬到 Facade 里面了吗？没有什么大变化啊？**

没错，说的很对，表面上看就是把客户端的代码搬到 Facade 里面了，但实质是发生了变化的。请思考：Facade 到底位于何处呢？是位于客户端还是在由 A、B、C 模块组成的系统这边呢？

答案肯定是在系统这边，这有什么不一样吗？

当然有了，如果 Facade 在系统这边，那么它就相当于屏蔽了外部客户端和系统内部模块的交互，从而把 A、B、C 模块组合成为一个整体对外，不但方便了客户端的调用，而且封装了系统内部的细节功能。也就是说 Facade 与各个模块交互的过程已经是内部实现了。这样一来，如果今后调用模块的算法发生了变化，比如变化成要先调用 B，然后调用 A，那么只需要修改 Facade 的实现就可以了。

另外一个好处，Facade 的功能可以被很多个客户端调用，也就是说 Facade 可以实现功能的共享，也就是实现复用。同样的调用代码就只用在 Facade 里面写一次就好了，而不用在多个调用的地方重复写。

还有一个潜在的好处，对使用 Facade 的人员来说，Facade 大大节省了他们的学习成本，他们只需要了解 Facade 即可，无须再深入到子系统内部，去了解每个模块的细节，也不用和多个模块交互，从而使得开发简单，学习也容易。

3. 有外观，但是可以不使用

虽然有了外观，如果有需要，外部还是可以绕开 Facade，而直接调用某个具体模块的接口，这样就能实现兼顾组合功能和细节功能。比如在客户端就想要使用 A 模块的功能，那么就不需要使用 Facade，可以直接调用 A 模块的接口。

示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        AModuleApi a = new AModuleImpl();
        a.testA();
    }
}
```

直接调用

4. 外观提供了缺省的功能实现

现在的系统是越做越大、越来越复杂，对软件的要求也就越来越高。为了提高系统的可重用性，通常会把一个大的系统分成很多个子系统，再把一个子系统分成很多更小的子系统，一直分下去，分到一个一个小的模块，这样一来，子系统的重用性会得到加强，也更容易对子系统进行定制和使用。

但是这也带来一个问题，如果用户不需要对子系统进行定制，仅仅就是想要使用它们来完成一定的功能，那么使用起来会比较麻烦，需要跟多个模块交互。

外观对象就可以为用户提供一个简单的、缺省的实现，这个实现对大多数的用户来说都是已经足够了的。但是外观并不限制那些需要更多定制功能的用户，可以直接越过外观去访问内部模块的功能。

5. 外观模式的调用顺序示意图

外观模式的调用顺序示意如图 3.6 所示。

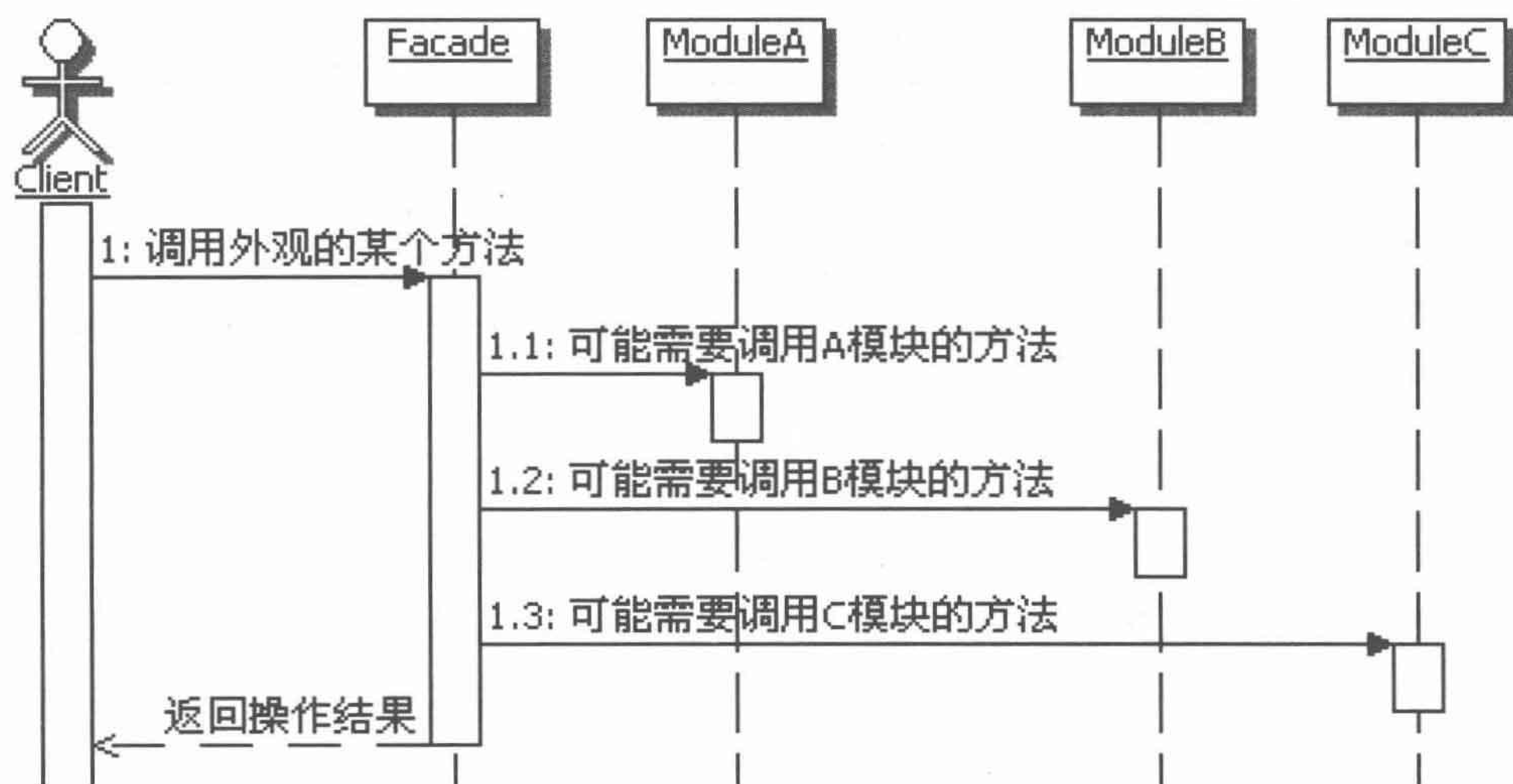


图 3.6　外观模式调用顺序示意图

3.3.2　外观模式的实现

1. Facade 的实现

对于一个子系统而言，外观类不需要很多，通常可以实现成为一个单例。

也可以直接把外观中的方法实现成为静态的方法，这样就可以不需要创建外观对象的实例而直接调用，这种实现相当于把外观类当成一个辅助工具类实现。简要的示例代码如下：

```
public class Facade {
    private Facade(){   }
    public static void test(){
        AModuleApi a = new AModuleImpl();
        a.testA();
        BModuleApi b = new BModuleImpl();
        b.testB();
        CModuleApi c = new CModuleImpl();
        c.testC();
    }
}
```

2. Facade 可以实现成为 interface

虽然 Facade 通常直接实现成为类，但是也可以把 Facade 实现成为真正的 interface。只是这样会增加系统的复杂程度，因为这样会需要一个 Facade 的实现，还需要一个来获取 Facade 接口对象的工厂。此时的结构如图 3.7 所示。

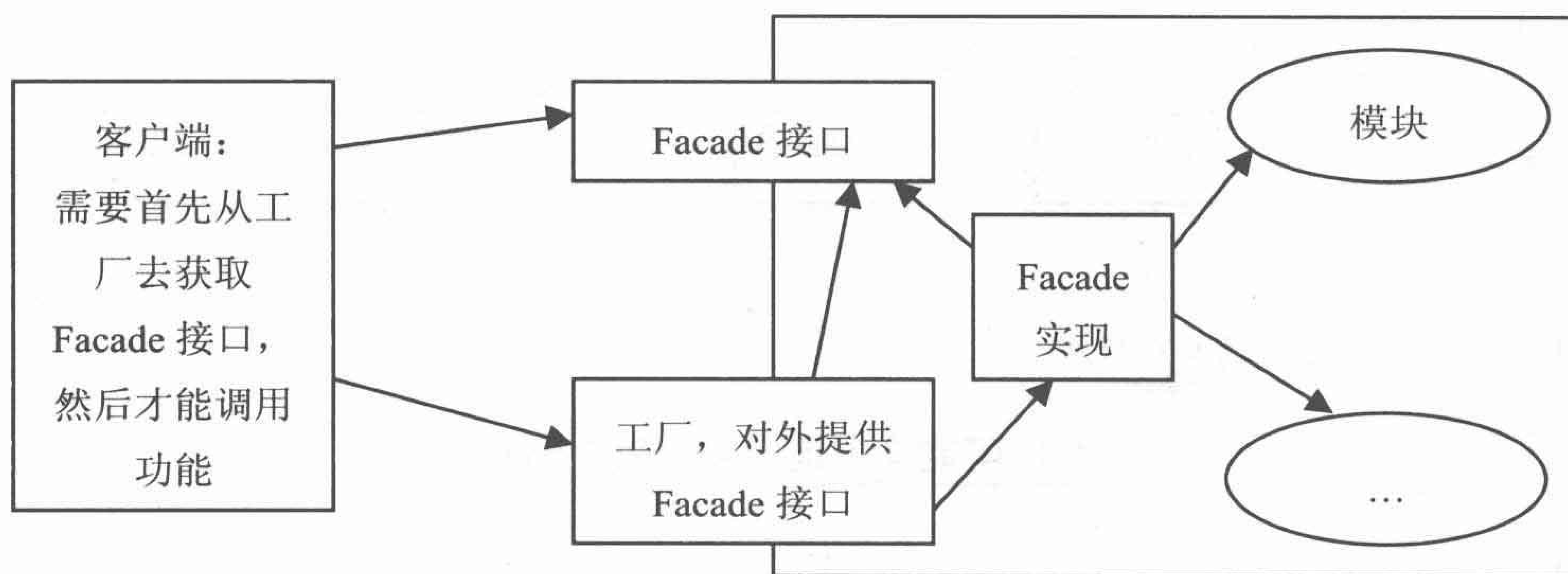


图 3.7　外观实现成为接口的结构示意图

3. Facade 实现成为 interface 的附带好处

如果把 Facade 实现成为接口，还附带一个功能，就是能够有选择性地暴露接口的方法，尽量减少模块对子系统外提供的接口方法。

> **提示**　**换句话说，一个模块的接口中定义的方法可以分成两部分，一部分是给子系统外部使用的，一部分是子系统内部的模块间相互调用时使用的。有了 Facade 接口，那么用于子系统内部的接口功能就不用暴露给子系统的外部了。**

比如，定义如下的 A、B、C 模块的接口：

```
public interface AModuleApi {
public void a1();        对子系统外部
```

```
public void a2();
public void a3();
}
```

这些方法是用在子系统内部，与B、C 模块交互用

同理定义 B、C 模块的接口

```
public interface BModuleApi {
//对子系统外部
    public void b1();
    //子系统内部使用
    public void b2();
    //子系统内部使用
    public void b3();
}
public interface CModuleApi {
    //对子系统外部
    public void c1();
    //子系统内部使用
    public void c2();
    //子系统内部使用
    public void c3();
}
```

定义好各个模块的接口，接下来定义 Facade 的接口：

```
public interface FacadeApi {
public void a1();
public void b1();
public void c1();

public void test();
}
```

这些是 A、B、C 模块对子系统外的接口，这样外部就不需要知道 A、B、C 模块的存在，只需要知道 Facade 接口就行了。

这是对外提供的组合方法，跟前面例子中的 Facade 类里面的方法一样。

这样定义 Facade 的话，外部只需要有 Facade 接口，就不再需要其他的接口了，可以有效地屏蔽内部的细节，免得客户端去调用 A 模块的接口时，发现一些不需要它知道的接口，将会造成“接口污染”。

比如 a2、a3 方法就不需要让客户端知道，否则既暴露了内部的细节，又让客户端迷惑。对客户端来说，他可能还要去思考 a2、a3 方法用来干什么呢？其实 a2、a3 方法是对内部模块之间交互的，原本就不是对子系统外部的，所以干脆就不要让客户端知道。

4. Facade 的方法实现

Facade 的方法实现中，一般是负责把客户端的请求转发给子系统内部的各个模块进

行处理，Facade 的方法本身并不进行功能的处理，Facade 的方法实现只是实现一个功能的组合调用。

当然在 Facade 中实现一个逻辑处理也并不是不可以的，但是不建议这样做，因为这不是 Facade 的本意，也超出了 Facade 的边界。

3.3.3 外观模式的优缺点

外观模式有如下优点。

- 松散耦合

  外观模式松散了客户端与子系统的耦合关系，让子系统内部的模块能更容易扩展和维护。
- 简单易用

  外观模式让子系统更加易用，客户端不再需要了解子系统内部的实现，也不需要跟众多子系统内部的模块进行交互，只需要跟外观交互就可以了，相当于外观类为外部客户端使用子系统提供了一站式服务。
- 更好地划分访问的层次

  通过合理使用 Facade，可以帮助我们更好地划分访问的层次。有些方法是对系统外的，有些方法是系统内部使用的。把需要暴露给外部的功能集中到外观中，这样既方便客户端使用，也很好地隐藏了内部的细节。

外观模式有如下缺点。

过多的或者是不太合理的 Facade 也容易让人迷惑。到底是调用 Facade 好呢，还是直接调用模块好。

3.3.4 思考外观模式

1. 外观模式的本质

> **外观模式的本质是**：封装交互，简化调用。

Facade 封装了子系统外部和子系统内部多个模块的交互过程，从而简化了外部的调用。通过外观，子系统为外部提供一些高层的接口，以方便它们的使用。

2. 对设计原则的体现

外观模式很好地体现了“最少知识原则”。

如果不使用外观模式，客户端通常需要和子系统内部的多个模块交互，也就是说客户端会有很多的朋友，客户端和这些模块之间都有依赖关系，任意一个模块的变动都可能会引起客户端的变动。

使用外观模式后，客户端只需要和外观类交互，也就是说客户端只有外观类这一个

朋友，客户端就不需要去关心子系统内部模块的变动情况了，客户端只是和这个外观类有依赖关系。

这样一来，客户端不但简单，而且这个系统会更有弹性。当系统内部多个模块发生变化的时候，这个变化可以被这个外观类吸收和消化，并不需要影响到客户端，换句话说就是：可以在不影响客户端的情况下，实现系统内部的维护和扩展。

3. 何时选用外观模式

建议在如下情况时选用外观模式。

- 如果你希望为一个复杂的子系统提供一个简单接口的时候，可以考虑使用外观模式。使用外观对象来实现大部分客户需要的功能，从而简化客户的使用。
- 如果想要让客户程序和抽象类的实现部分松散耦合，可以考虑使用外观模式，使用外观对象来将这个子系统与它的客户分离开来，从而提高子系统的独立性和可移植性。
- 如果构建多层结构的系统，可以考虑使用外观模式，使用外观对象作为每层的入口，这样可以简化层间调用，也可以松散层次之间的依赖关系。

3.3.5　相关模式

- 外观模式和中介者模式

  这两个模式非常类似，但是却有本质的区别。

  中介者模式主要用来封装多个对象之间相互的交互，多用在系统内部的多个模块之间；而外观模式封装的是单向的交互，是从客户端访问系统的调用，没有从系统中来访问客户端的调用。

  在中介者模式的实现里面，是需要实现具体的交互功能的；而外观模式的实现里面，一般是组合调用或是转调内部实现的功能，通常外观模式本身并不实现这些功能。

  中介者模式的目的主要是松散多个模块之间的耦合，把这些耦合关系全部放到中介者中去实现；而外观模式的目的是简化客户端的调用，这点和中介者模式也不同。
- 外观模式和单例模式

  通常一个子系统只需要一个外观实例，所以外观模式可以和单例模式组合使用，把 Facade **类实现成为单例**。当然，也可以跟前面示例的那样，把外观类的构造方法私有化，然后把提供给客户端的方法实现成为静态的。
- 外观模式和抽象工厂模式

  外观模式的外观类通常需要和系统内部的多个模块交互，每个模块一般都有自己的接口，**所以在外观类的具体实现里面**，需要获取这些接口，然后组合这些接口来完成客户端的功能。

那么怎么获取这些接口呢？就可以和抽象工厂一起使用，外观类通过抽象工厂来获取所需要的接口，而抽象工厂也可以把模块内部的实现对 Facade 进行屏蔽，也就是说 Facade 也仅仅只是知道它从模块中获取它需要的功能，模块内部的细节，Facade 也不知道。

第 4 章　适配器模式（Adapter）

4.1 场景问题

4.1.1 装配电脑的例子

1. 旧的硬盘和电源

小李有一台老的台式电脑，硬盘实在是太小了，仅仅40GB，但是除了这个问题外，整机性能还不错，废弃不用太可惜了，于是决定去加装一块新的硬盘。

在装机公司为小李的电脑加装新硬盘的时候，小李也在旁边观看，顺便了解一些硬件知识。很快，装机人员把两块硬盘都安装好了，细心的小李发现，这两块硬盘的连接方式是不一样的。

经过装机人员的耐心讲解，小李搞清楚了它们的不同。以前的硬盘是串口的，如图4.1所示，电脑电源如图4.2所示，在连接电源的时候是直接连接。



图4.1 旧的硬盘

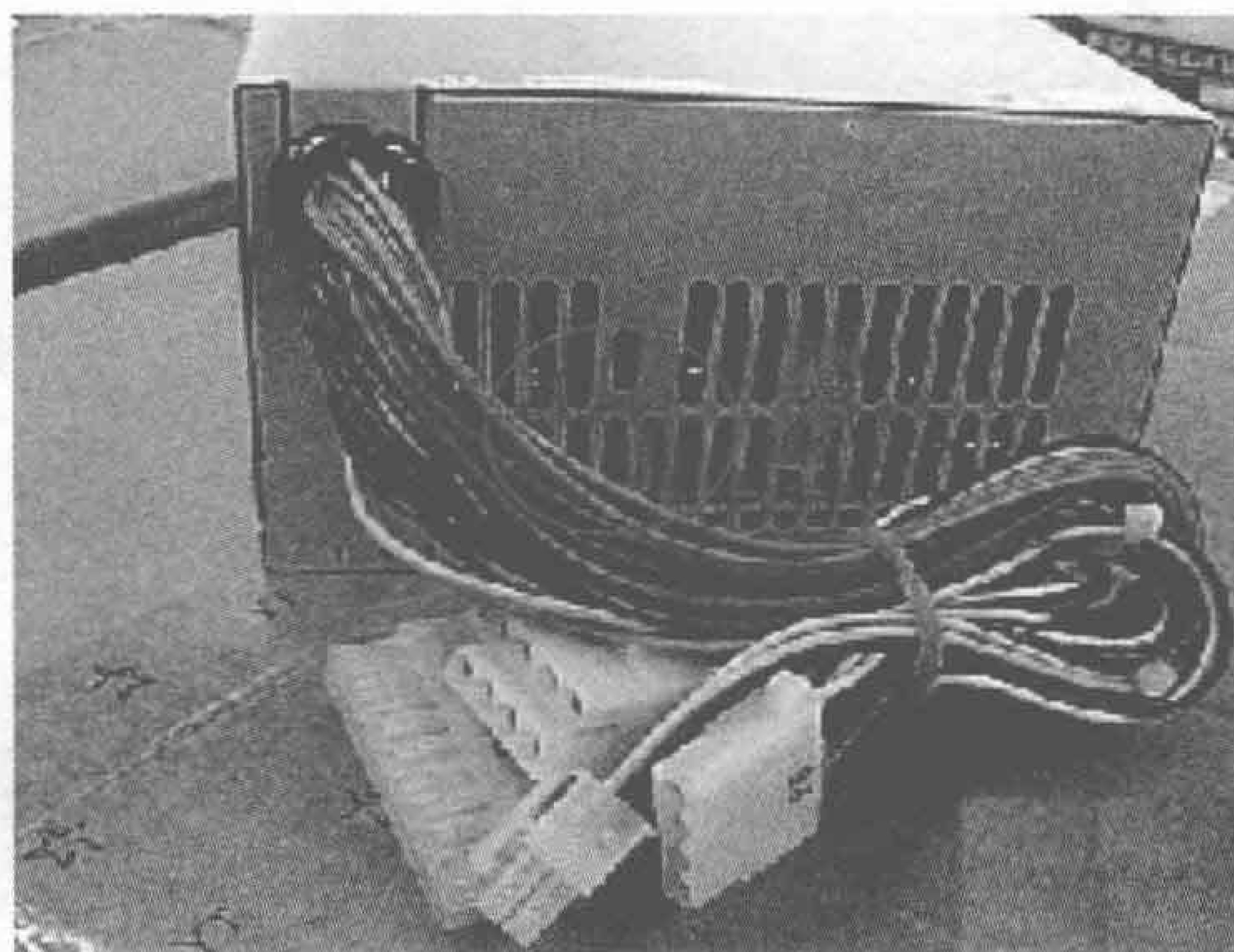


图4.2 电脑电源

2. 加入新的硬盘

现在的新硬盘是并口的，如图4.3所示，电源的输出口无法直接连接到新的硬盘上了。于是就有了转接线，一边和电源的输出口连接，一边和新的硬盘电源输入口连接，解决了电源输出接口和硬盘输入接口不匹配的问题，如图4.4所示。

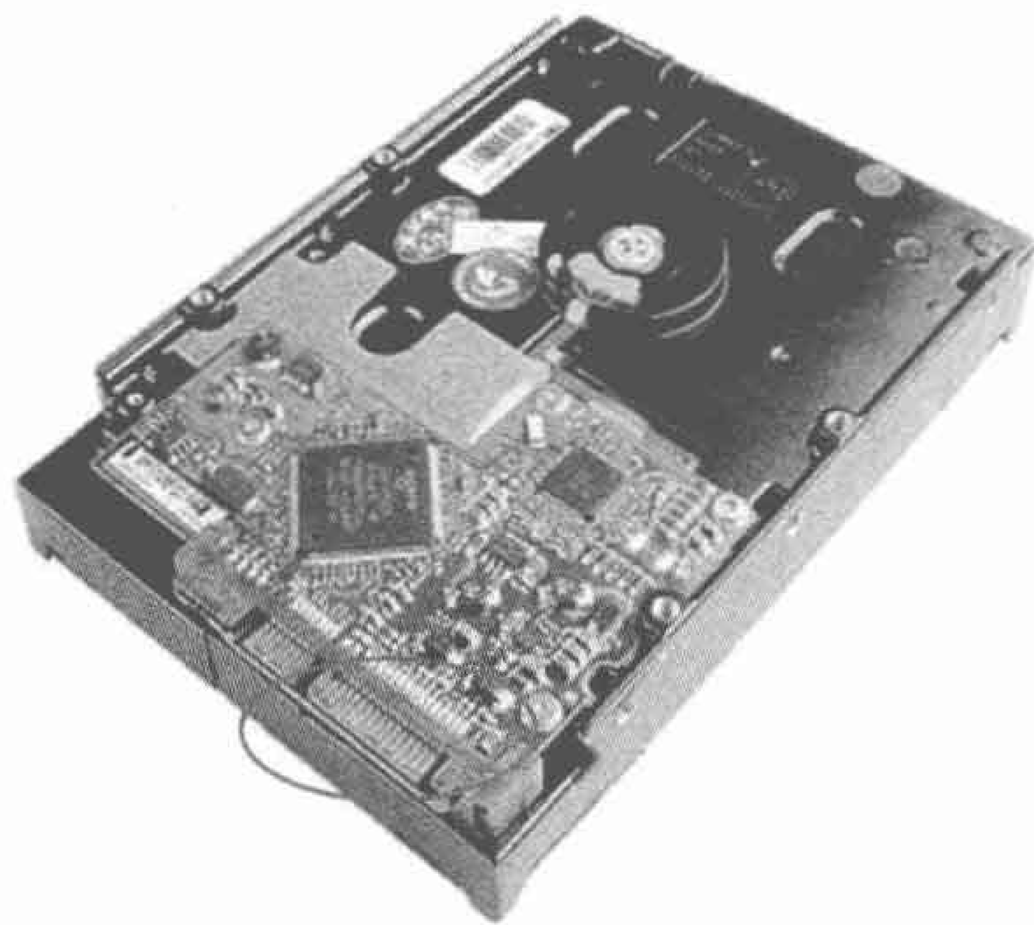


图4.3 新的硬盘

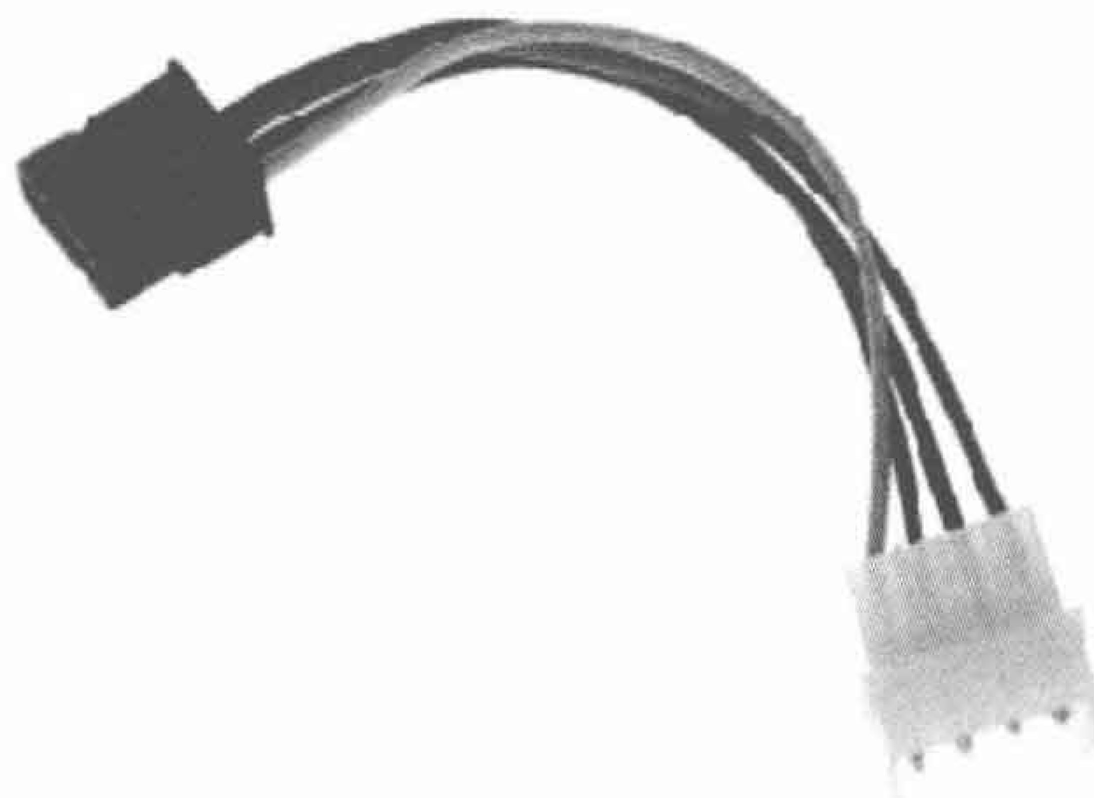


图4.4 电源转接线

3. 有何问题

如果把上面的问题抽象一下，用对象来描述，那就是：有一个电源类和旧的硬盘类

配合工作得很好，现在又有了一个新的硬盘类，现在想让新的硬盘类和电源类也配合使用，但是发现它们的接口无法匹配，问题就产生了：**如何让原有的电源类的接口能够适应新的硬盘类的电源接口的需要呢？**

4. 如何解决

解决方法是采用一个转接线类，转接线可以把电源的接口适配成为新的硬盘所需要的接口，那么这个转接线类就类似本章的主角——适配器（Adapter）。

4.1.2　同时支持数据库和文件的日志管理

看了上面这个例子，估计对适配器模式有一点感觉了。这是一个在生活中常见的例子，类似的例子很多，比如，各种管道的转接头、不同制式的插座等。但是这种例子只能帮助大家理解适配器模式的功能，跟实际的应用系统开发总是有一些差距，让人感觉到好像是理解了模式的功能，但是一到真实的系统开发中，就不知道如何使用这个模式了，有些隔靴搔痒的感觉。因此，下面还是以实际系统中的例子来讲述，以帮助大家真正理解和应用适配器模式。

考虑一个记录日志的应用，由于用户对日志记录的要求很高，使得开发人员不能简单地采用一些已有的日志工具或日志框架来满足用户的要求，而需要按照用户的要求重新开发新的日志管理系统。当然这里不可能完全按照实际系统那样去完整实现，只是抽取跟适配器模式相关的部分来讲述。

1. 日志管理第一版

在第一版的时候，用户要求日志以文件的形式记录。开发人员遵照用户的要求，对日志文件的存取实现如下。

（1）先简单定义日志对象，也就是描述日志的对象模型。由于这个对象需要被写入文件中，因此这个对象需要序列化。示例代码如下：

```
/**
 * 日志数据对象
 */
public class LogModel
    /**
     * 日志编号
     */
    private String logId;
    /**
     * 操作人员
     */
    private String operateUser;
    /**
     * 操作时间，以 yyyy-MM-dd HH:mm:ss 的格式记录
     */
```

```
    private String operateTime;
    /**
     * 日志内容
     */
    private String logContent;

    public String getLogId() {
        return logId;
    }
    public void setLogId(String logId) {
        this.logId = logId;
    }
    public String getOperateUser() {
        return operateUser;
    }
    public void setOperateUser(String operateUser) {
        this.operateUser = operateUser;
    }
    public String getOperateTime() {
        return operateTime;
    }
    public void setOperateTime(String operateTime) {
        this.operateTime = operateTime;
    }
    public String getLogContent() {
        return logContent;
    }
    public void setLogContent(String logContent) {
        this.logContent = logContent;
    }
    public String toString(){
        return "logId="+logId+",operateUser="+operateUser
         +",operateTime="+operateTime+",logContent="+logContent;
    }
}
```


（2）接下来定义一个操作日志文件的接口。示例代码如下：

```
/**
 * 日志文件操作接口
 */
public interface LogFileOperateApi {
```

```
    /**
     * 读取日志文件，从文件里面获取存储的日志列表对象
     * @return 存储的日志列表对象
     */
    public List<LogModel> readLogFile();
    /**
     * 写日志文件，把日志列表写出到日志文件中去
     * @param list 要写到日志文件的日志列表
     */
    public void writeLogFile(List<LogModel> list);
}
```

（3）实现日志文件的存取。现在的实现也很简单，就是读写文件。示例代码如下：

```
/**
 * 实现对日志文件的操作
 */
public class LogFileOperate implements LogFileOperateApi{
    /**
     * 日志文件的路径和文件名称，默认是当前项目根下的 AdapterLog.log
     */
    private String logFilePathName = "AdapterLog.log";
    /**
     * 构造方法，传入文件的路径和名称
     * @param logFilePathName 文件的路径和名称
     */
    public LogFileOperate(String logFilePathName) {
        //先判断是否传入了文件的路径和名称，如果是，
        //就重新设置操作的日志文件的路径和名称
        if(logFilePathName!=null &&
                            logFilePathName.trim().length()>0){
            this.logFilePathName = logFilePathName;
        }
    }
    public  List<LogModel> readLogFile() {
        List<LogModel> list = null;
        ObjectInputStream oin = null;
        try {
            File f = new File(logFilePathName);
            if(f.exists()){
                oin = new ObjectInputStream(
                        new BufferedInputStream(
```

```
                        new FileInputStream(f))
                );
                list = (List<LogModel>)oin.readObject();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            try {
                if(oin!=null){
                    oin.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return list;
    }

    public void writeLogFile(List<LogModel> list){
        File f = new File(logFilePathName);
        ObjectOutputStream oout = null;
        try {
            oout = new ObjectOutputStream(
                        new BufferedOutputStream(
                          new FileOutputStream(f))
            );
            oout.writeObject(list);
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                oout.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

（4）写个客户端来测试一下，看看好用不。示例代码如下：

```
public class Client {
```

```
    public static void main(String[] args) {
        //准备日志内容，也就是测试的数据
        LogModel lm1 = new LogModel();
        lm1.setLogId("001");
        lm1.setOperateUser("admin");
        lm1.setOperateTime("2010-03-02 10:08:18");
        lm1.setLogContent("这是一个测试");

        List<LogModel> list = new ArrayList<LogModel>();
        list.add(lm1);
        //创建操作日志文件的对象
        LogFileOperateApi api = new LogFileOperate("");
        //保存日志文件
        api.writeLogFile(list);

        //读取日志文件的内容
        List<LogModel> readLog = api.readLogFile();
        System.out.println("readLog="+readLog);
    }
}
```

测试的结果如下：

```
readLog=[logId=001,operateUser=admin,operateTime=2010-03-02
10:08:18,logContent=这是一个测试]
```

至此就简单的实现了用户的要求，把日志保存到文件中，并能从文件中把日志内容读取出来，进行管理。

看上去很容易，对吧，别慌，接着来。

2. 日志管理第二版

用户使用日志管理第一版一段时间后，开始考虑升级系统，决定要采用数据库来管理日志。很快，按照数据库的日志管理也实现出来了，并定义了日志管理的操作接口，主要是针对日志的增删改查方法。接口的示例代码如下：

```
/**
 * 定义操作日志的应用接口，为了示例的简单，只是简单地定义了增删改查的方法
 */
public interface LogDbOperateApi {
    /**
     * 新增日志
     * @param lm 需要新增的日志对象
     */
    public void createLog(LogModel lm);
```

```
    /**
     * 修改日志
     * @param lm 需要修改的日志对象
     */
    public void updateLog(LogModel lm);
    /**
     * 删除日志
     * @param lm 需要删除的日志对象
     */
    public void removeLog(LogModel lm);
    /**
     * 获取所有的日志
     * @return 所有的日志对象
     */
    public List<LogModel> getAllLog();
}
```

对于使用数据库来保存日志的实现，这里就不去涉及了，总之知道有这么一个实现就可以了。

客户提出了新的要求，能不能让日志管理第二版实现同时支持数据库存储和文件存储两种方式？

4.1.3 有何问题

有朋友可能会想，这有什么困难的呢，两种实现方式不是都已经实现了的吗，合并起来不就可以了？

问题就在于，现在的业务是使用的第二版的接口，直接使用第二版新加入的实现是没有问题的，第二版新加入了保存日志到数据库中；但是对于已有的实现方式，也就是在第一版中采用的文件存储的方式，它的操作接口和第二版不一样，这就导致现在的客户端无法以同样的方式来直接使用第一版的实现，如图 4.5 所示。

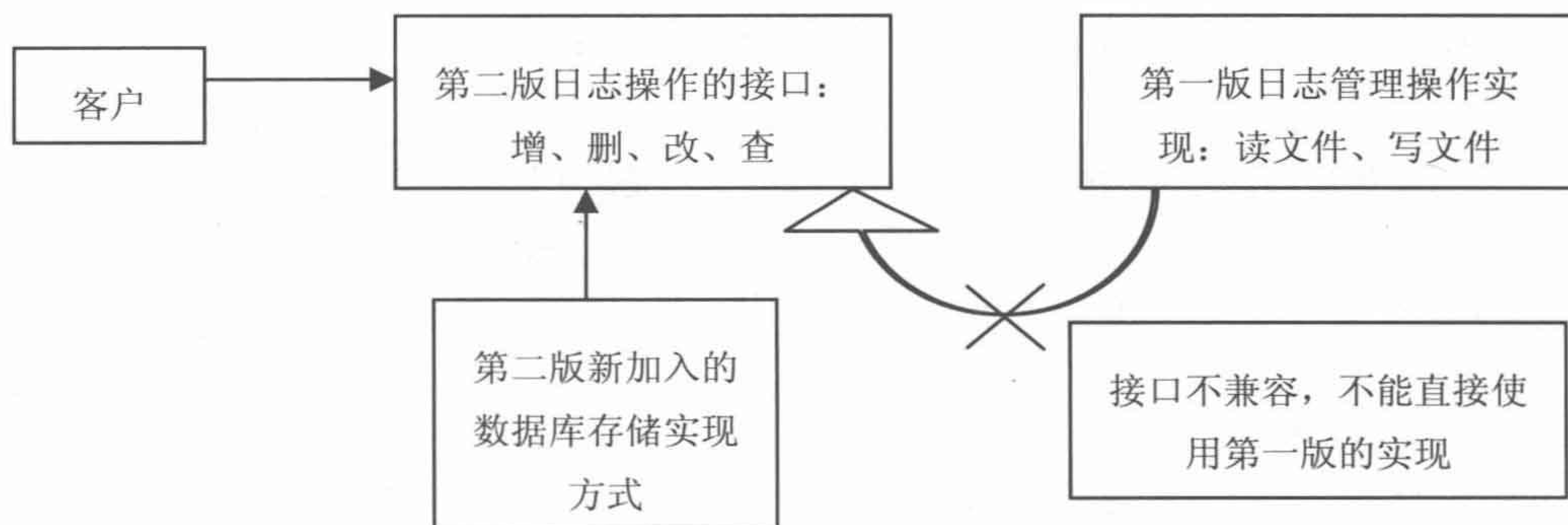


图 4.5　无法兼容第一版的接口示意图

这就意味着，要想同时支持文件和数据库存储两种方式，需要再额外地做一些工作，才可以让第一版的实现适应新的业务需要。

可能有朋友会想，干脆按照第二版的接口要求重新实现一个文件操作的对象不就可以了吗，这样做确实可以，但是何必要重新做已经完成的功能呢？应该想办法复用，而不是重新实现。

一种很容易想到的方式是直接修改已有的第一版的代码。这种方式是不太好的，如果直接修改第一版的代码，那么可能会导致其他依赖于这些实现的应用不能正常运行，再说，有可能第一版和第二版的开发公司是不一样的，在第二版实现的时候，根本拿不到第一版的源代码。

那么该如何来实现呢？

4.2 解决方案

4.2.1 使适配器模式来解决问题

用来解决上述问题的一个合理的解决方案就是适配器模式。那么什么是适配器模式呢？

1. 适配器模式的定义

将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

2. 应用适配器模式来解决问题的思路

仔细分析上面的问题，问题的根源在于接口的不兼容，功能是基本实现了的，也就是说，只要想办法让两边的接口匹配起来，就可以复用第一版的功能了。

按照适配器模式的实现方式，可以定义一个类来实现第二版的接口，然后在内部实现的时候，转调第一版已经实现了的功能，这样就可以通过对象组合的方式，既复用了第一版已有的功能，同时又在接口上满足了第二版调用的要求。

完成上述工作的这个类就是适配器。

4.2.2 适配器模式的结构和说明

适配器模式的结构如图 4.6 所示。

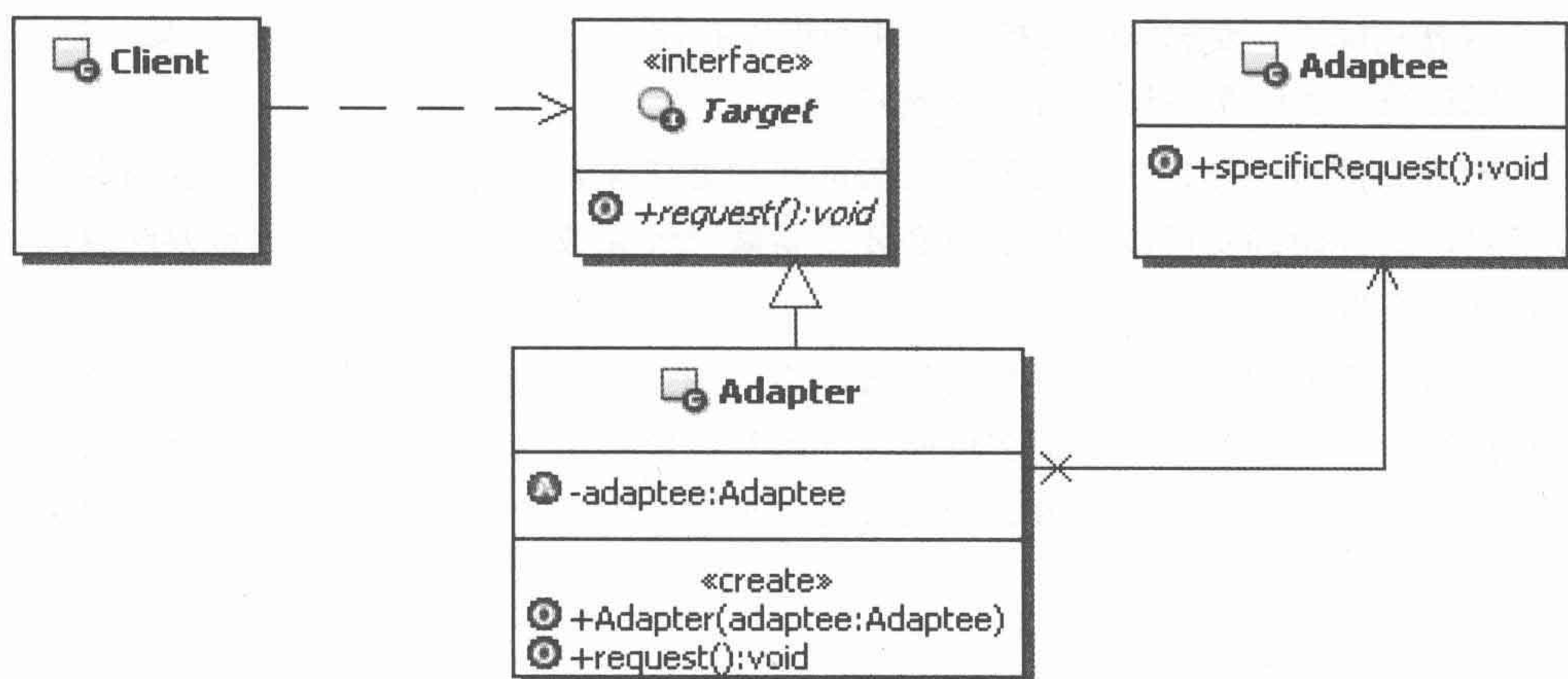


图 4.6　适配器模式的结构图

- Client：客户端，调用自己需要的领域接口 Target。
- Target：定义客户端需要的跟特定领域相关的接口。
- Adaptee：已经存在的接口，通常能满足客户端的功能要求，但是接口与客户端要求的特定领域接口不一致，需要被适配。
- Adapter：适配器，把 Adaptee 适配成为 Client 需要的 Target。

4.2.3　适配器模式示例代码

（1）先看看 Target 接口定义的示例代码如下：

```
/**
 * 定义客户端使用的接口，与特定领域相关
 */
public interface Target {
    /**
     * 示意方法，客户端请求处理的方法
     */
    public void request();
}
```

（2）再看看需要被适配的对象定义。示例代码如下：

```
/**
 * 已经存在的接口，这个接口需要被适配
 */
public class Adaptee {
    /**
     * 示意方法，原本已经存在，已经实现的方法
     */
    public void specificRequest() {
```

```
        //具体的功能处理
    }
}
```

（3）下面是适配器对象的基本实现。示例代码如下：

```
/**
 * 适配器
 */
public class Adapter implements Target {
    /**
     * 持有需要被适配的接口对象
     */
    private Adaptee adaptee;
    /**
     * 构造方法，传入需要被适配的对象
     * @param adaptee 需要被适配的对象
     */
    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    public void request() {
        //可能转调已经实现了的方法，进行适配
        adaptee.specificRequest();
    }
}
```

（4）再来看看使用适配器客户端的示例代码如下：

```
/**
 * 使用适配器的客户端
 */
public class Client {
    public static void main(String[] args) {
        //创建需要被适配的对象
        Adaptee adaptee = new Adaptee();
        //创建客户端需要调用的接口对象
        Target target = new Adapter(adaptee);
        //请求处理
        target.request();
    }
}
```

4.2.4 使用适配器模式来实现示例

要使用适配器模式来实现示例，关键是要实现适配器对象。它需要实现第二版的接口，但是在内部实现的时候，需要调用第一版已经实现的功能。也就是说，第二版的接口就相当于适配器模式中的 Target 接口，而第一版已有的实现就相当于适配器模式中的 Adaptee 对象。

（1）把适配器简单的实现出来，示意一下。示例代码如下：

```
/**
 * 适配器对象，将记录日志到文件的功能适配成第二版需要的增删改查功能
 */
public class Adapter implements LogDbOperateApi{
    /**
     * 持有需要被适配的接口对象
     */
    private LogFileOperateApi adaptee;
    /**
     * 构造方法，传入需要被适配的对象
     * @param adaptee 需要被适配的对象
     */
    public Adapter(LogFileOperateApi adaptee) {
        this.adaptee = adaptee;
    }

    public void createLog(LogModel lm) {
        //1：先读取文件的内容
        List<LogModel> list = adaptee.readLogFile();
        //2：加入新的日志对象
        list.add(lm);
        //3：重新写入文件
        adaptee.writeLogFile(list);
    }
    public List<LogModel> getAllLog() {
        return adaptee.readLogFile();
    }
    public void removeLog(LogModel lm) {
        //1：先读取文件的内容
        List<LogModel> list = adaptee.readLogFile();
        //2：删除相应的日志对象
        list.remove(lm);
        //3：重新写入文件
```

```
        adaptee.writeLogFile(list);
    }
    public void updateLog(LogModel lm) {
        //1：先读取文件的内容
        List<LogModel> list = adaptee.readLogFile();
        //2：修改相应的日志对象
        for(int i=0;i<list.size();i++){
            if(list.get(i).getLogId().equals(lm.getLogId())){
                list.set(i, lm);
                break;
            }
        }
        //3：重新写入文件
        adaptee.writeLogFile(list);
    }
}
```

（2）此时的客户端也需要一些改变。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备日志内容，也就是测试的数据
        LogModel lm1 = new LogModel();
        lm1.setLogId("001");
        lm1.setOperateUser("admin");
        lm1.setOperateTime("2010-03-02 10:08:18");
        lm1.setLogContent("这是一个测试");
        List<LogModel> list = new ArrayList<LogModel>();
        list.add(lm1);
        //创建操作日志文件的对象
        LogFileOperateApi logFileApi = new LogFileOperate("");

        //创建新版操作日志的接口对象
        LogDbOperateApi api = new Adapter(logFileApi);

        //保存日志文件
        api.createLog(lm1);
        //读取日志文件的内容
        List<LogModel> allLog = api.getAllLog();
        System.out.println("allLog="+allLog);
    }
}
```

运行上述代码，测试其是否能满足要求。

（3）下面总结一下这个思路。

① 原有文件存取日志的方式，运行得很好，如图 4.7 所示。

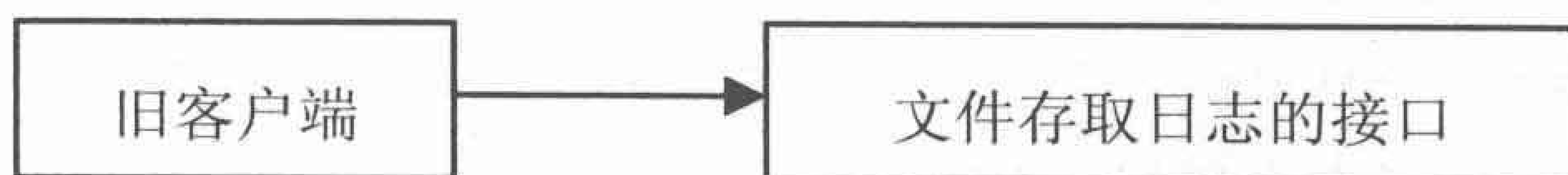


图 4.7　原有文件存取日志的方式

② 现在有了新的基于数据库的实现，新的实现有自己的接口，如图 4.8 所示。

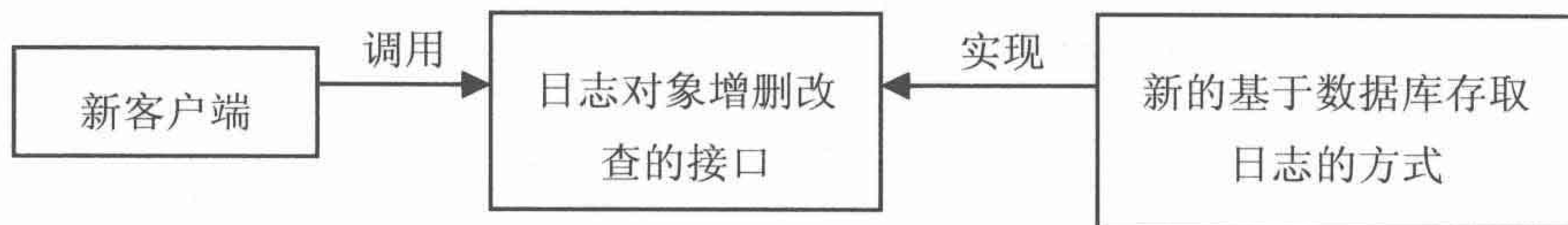


图 4.8　新的基于数据库的实现

③ 现在想要在第二版的实现里面，能够同时兼容第一版的功能，那么就应有一个类来实现第二版的接口，然后在这个类里面去调用已有的功能实现，这个类就是适配器，如图 4.9 所示。

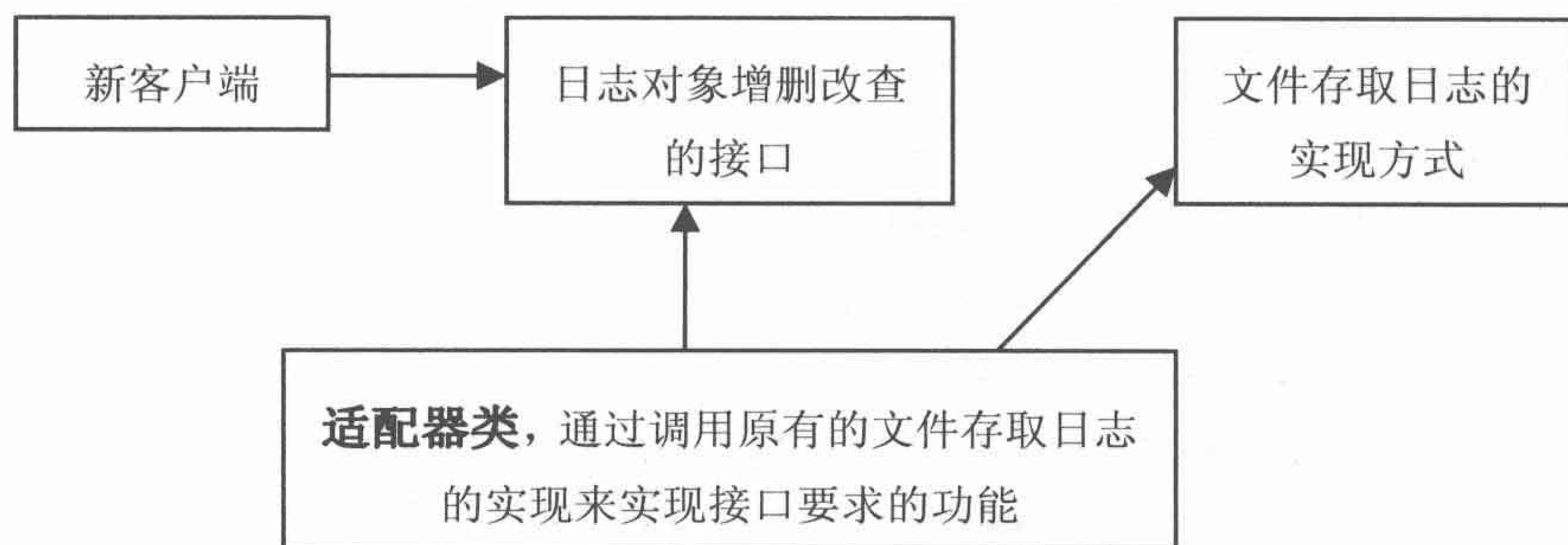


图 4.9　加入适配器的实现结构示意图

上面是分步的思路，下面来看一下前面示例的整体结构，如图 4.10 所示。

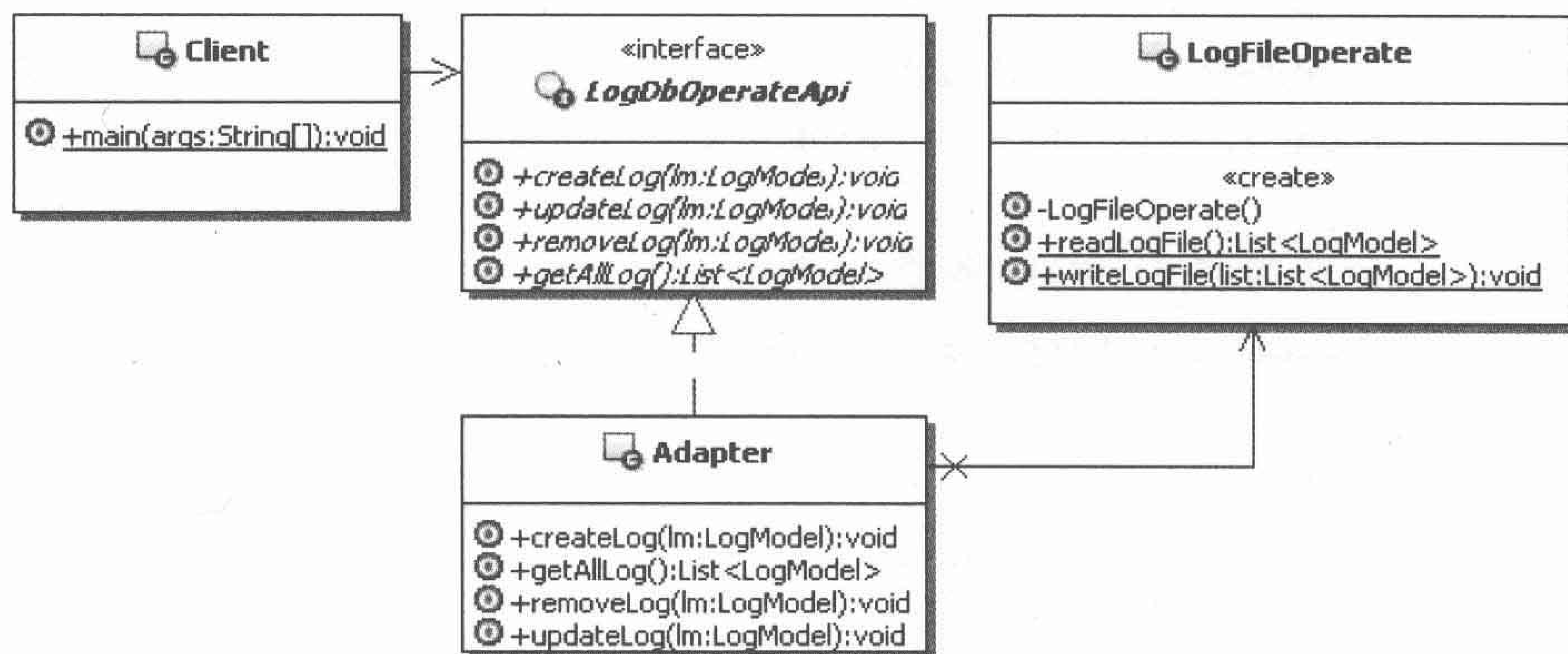


图 4.10　适配器实现示例的结构示意图

如同上面的例子，原本新的日志操作接口不能和旧的文件实现一起工作，但是经过适配器适配后，新的日志操作接口就可以和旧的文件实现日志存储一起工作了。

4.3 模式讲解

4.3.1 认识适配器模式

1. 模式的功能

适配器模式的主要功能是进行转换匹配，目的是复用已有的功能，而不是来实现新的接口。也就是说，客户端需要的功能应该是已经实现好了的，不需要适配器模式来实现，适配器模式主要负责把不兼容的接口转换成客户端期望的样子就可以了。

但这并不是说，在适配器里面就不能实现功能。适配器里面可以实现功能，称这种适配器为智能适配器。再说了，在接口匹配和转换的过程中，也有可能需要额外实现一定的功能，才能够转换过来，比如需要调整参数以进行匹配等。

2. Adaptee 和 Target 的关系

适配器模式中被适配的接口 Adaptee 和适配成为的接口 Target 是没有关联的，也就是说，Adaptee 和 Target 中的方法既可以相同，也可以不同。极端情况下两个接口里面的方法可能是完全不同的，当然这种情况下也可以完全相同。

这里所说的相同和不同，是指方法定义的名称、参数列表、返回值，以及方法本身的功能都可以相同或不同。

3. 对象组合

根据前面的实现，你会发现，适配器的实现方式其实是依靠对象组合的方式。通过给适配器对象组合被适配的对象，然后当客户端调用 Target 的时候，适配器会把相应的功能委托给被适配的对象去完成。

4. 适配器模式的调用顺序示意图

适配器模式的调用顺序如图 4.11 所示。

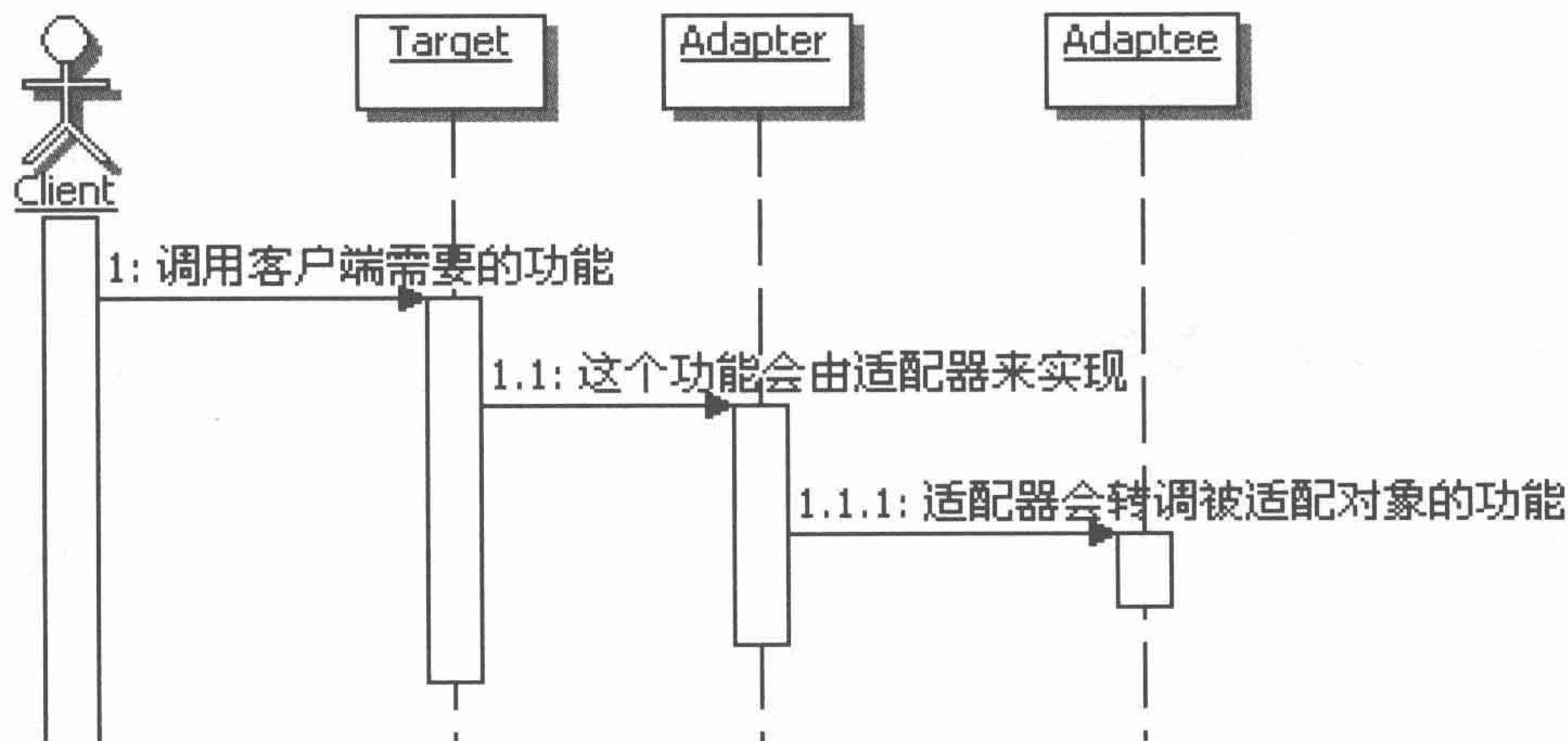


图 4.11　适配器模式的调用顺序示意图

4.3.2 适配器模式的实现

1. 适配器的常见实现

在实现适配器的时候，适配器通常是一个类，一般会让适配器类去实现 Target 接口，然后在适配器的具体实现里面调用 Adaptee。也就是说适配器通常是一个 Target 类型，而不是 Adaptee 类型。如同前面的例子演示的那样。

2. 智能适配器

在实际开发中，适配器也可以实现一些 Adaptee 没有实现，但是在 Target 中定义的功能。这种情况就需要在适配器的实现里面，加入新功能的实现。这种适配器被称为智能适配器。

如果要使用智能适配器，一般新加入功能的实现会用到很多 Adaptee 的功能，相当于利用 Adaptee 的功能来实现更高层的功能。当然也可以完全实现新加入的功能，和已有的功能都不相关，变相地扩展了功能。

3. 适配多个 Adaptee

适配器在适配的时候，可以适配多个 Adaptee，也就是说实现某个新的 Target 的功能的时候，需要调用多个模块的功能，适配多个模块的功能才能满足新接口的要求。

4. 适配器 Adapter 实现的复杂程度

适配器 Adapter 实现的复杂程度取决于 Target 和 Adaptee 的相似程度。

如果相似程度很高，比如只有方法名称不一样，那么 Adapter 只需要简单地转调一下接口就可以了。

如果相似程度低，比如两边接口的方法所定义的功能完全不一样，在 Target 中定义的一个方法，可能在 Adaptee 中定义了三个更小的方法，那么这个时候在实现 Adapter 的时候，就需要组合调用了。

5. 缺省适配

缺省适配的意思是，为一个接口提供缺省实现。有了它，就不用直接去实现接口，而是采用继承这个缺省适配对象，从而让子类可以有选择地去覆盖实现需要的方法，对于不需要的方法，使用缺省适配的方法就可以了。

4.3.3 双向适配器

适配器也可以实现双向的适配，前面我们讲的都是把 Adaptee 适配成为 Target，其实也可以把 Target 适配成为 Adaptee。也就是说这个适配器可以同时当作 Target 和 Adaptee 来使用。

继续前面讲述的例子。如果说由于某些原因，第一版和第二版会同时共存一段时间，比如第二版的应用还在不断调整中，也就是第二版还不够稳定。客户提出，希望在两版共存期间，主要还是使用第一版，同时希望第一版的日志也能记录到数据库中，也就是客户虽然操作的接口是第一版的日志接口，界面也是第一版的界面，但是可以使用第二

版的将日志记录到数据库的功能。

也就是说希望两版能实现双向的适配，结构如图 4.12 所示。

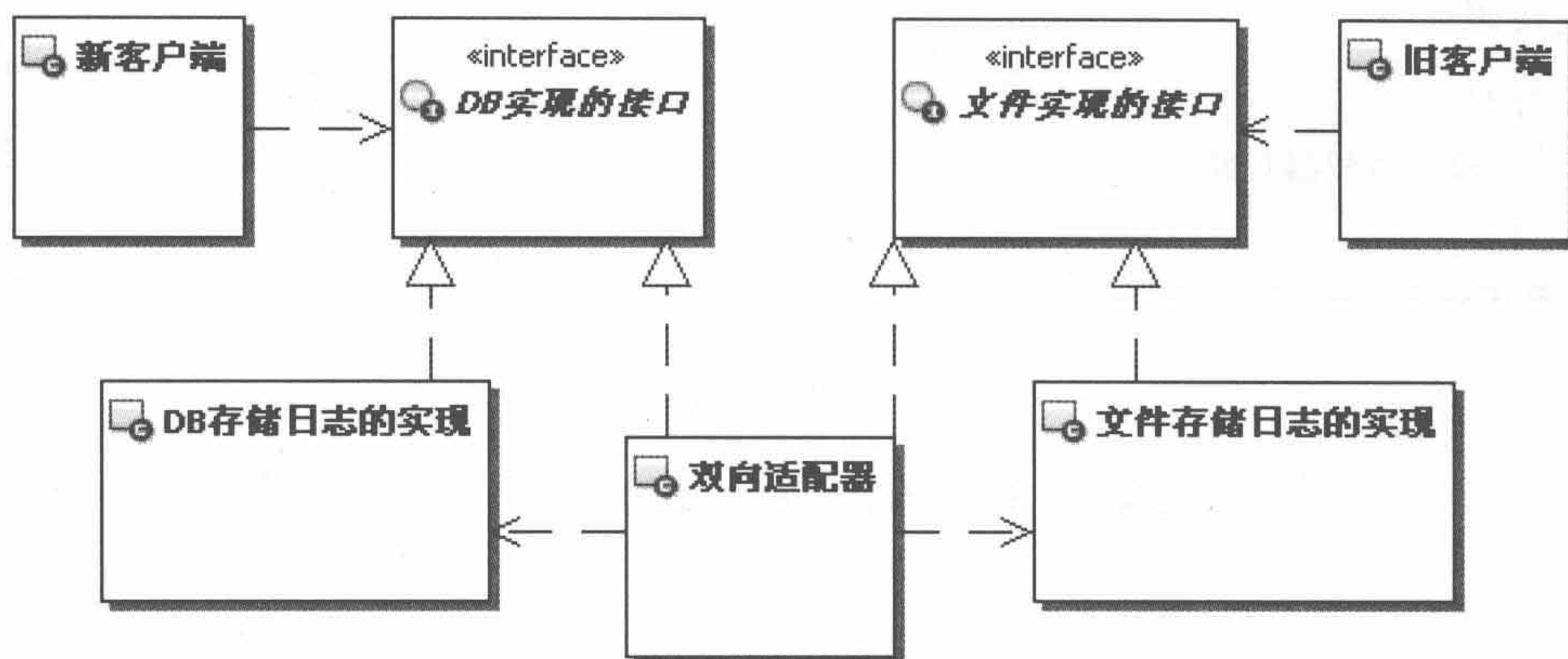


图 4.12　双向适配器示意图

下面用简单的代码示意一下，以利于大家理解。

这里只加了几个新的东西，一个是 DB 存储日志的实现，前面的例子中没有，因为直接被适配成使用文件存储日志的实现了；另外一个就是双向适配器，其实与把文件存储的方式适配成为 DB 实现的接口是一样的，只需要新加上把 DB 实现的功能适配成为文件实现的接口就可以了。

（1）先看看 DB 存储日志的实现。为了简单，这里不再真正地实现和数据库交互了，示意一下就可以了。示例代码如下：

```
/**
 * DB 存储日志的实现，为了简单，这里就不去真正地实现和数据库交互了，示意一下
 */
public class LogDbOperate implements LogDbOperateApi{
    public void createLog(LogModel lm) {
        System.out.println("now in LogDbOperate createLog,lm="+lm);
    }
    public List<LogModel> getAllLog() {
        System.out.println("now in LogDbOperate getAllLog");
        return null;
    }
    public void removeLog(LogModel lm) {
        System.out.println("now in LogDbOperate removeLog,lm="+lm);
    }
    public void updateLog(LogModel lm) {
        System.out.println("now in LogDbOperate updateLog,lm="+lm);
    }
}
```

（2）然后看看新的适配器的实现。

由于是双向的适配器，一个方向是：把新的 DB 实现的接口适配成为旧的文件操作需要的接口；另外一个方向是把旧的文件操作的接口适配成为新的 DB 实现需要的接口。示例代码如下：

```
/**
 * 双向适配器对象
 */
public class TwoDirectAdapter implements
                                LogDbOperateApi,LogFileOperateApi{
    /**
     * 持有需要被适配的文件存储日志的接口对象
     */
    private LogFileOperateApi fileLog;
    /**
     * 持有需要被适配的 DB 存储日志的接口对象
     */
    private LogDbOperateApi  dbLog;
    /**
     * 构造方法，传入需要被适配的对象
     * @param fileLog 需要被适配的文件存储日志的接口对象
     * @param dbLog 需要被适配的 DB 存储日志的接口对象
     */
    public TwoDirectAdapter(LogFileOperateApi fileLog
                            ,LogDbOperateApi dbLog) {
        this.fileLog = fileLog;
        this.dbLog = dbLog;
    }
    /*-----以下是把文件操作的方式适配成为 DB 实现方式的接口-----*/
    public void createLog(LogModel lm) {
        //1：先读取文件的内容
        List<LogModel> list = fileLog.readLogFile();
        //2：加入新的日志对象
        list.add(lm);
        //3：重新写入文件
        fileLog.writeLogFile(list);
    }
    public List<LogModel> getAllLog() {
        return fileLog.readLogFile();
    }
    public void removeLog(LogModel lm) {
        //1：先读取文件的内容
```

实现需要适配的两个接口

持有双向适配的日志接口对象

```
        List<LogModel> list = fileLog.readLogFile();
        //2：删除相应的日志对象
        list.remove(lm);
        //3：重新写入文件
        fileLog.writeLogFile(list);
    }
    public void updateLog(LogModel lm) {
        //1：先读取文件的内容
        List<LogModel> list = fileLog.readLogFile();
        //2：修改相应的日志对象
        for(int i=0;i<list.size();i++){
            if(list.get(i).getLogId().equals(lm.getLogId())){
                list.set(i, lm);
                break;
            }
        }
        //3：重新写入文件
        fileLog.writeLogFile(list);
    }
/*-----以下是把 DB 操作的方式适配成为文件实现方式的接口-----*/
    public List<LogModel> readLogFile() {
        return dbLog.getAllLog();
    }
    public void writeLogFile(List<LogModel> list) {
        //1：最简单的实现思路是先删除数据库中的数据
        //2：然后循环把现在的数据加入到数据库中
        for(LogModel lm : list){
            dbLog.createLog(lm);
        }
    }
}
```

（3）下面看看如何使用这个双向适配器。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备日志内容，也就是测试的数据
        LogModel lm1 = new LogModel();
        lm1.setLogId("001");
        lm1.setOperateUser("admin");
        lm1.setOperateTime("2010-03-02 10:08:18");
        lm1.setLogContent("这是一个测试");
```

```
        List<LogModel> list = new ArrayList<LogModel>();
        list.add(lm1);

        //创建操作日志文件的对象
        LogFileOperateApi fileLogApi = new LogFileOperate("");
        LogDbOperateApi dbLogApi = new LogDbOperate();

        //创建经过双向适配后的操作日志的接口对象
        LogFileOperateApi fileLogApi2 =
                new TwoDirectAdapter(fileLogApi,dbLogApi);
        LogDbOperateApi dbLogApi2 =
                new TwoDirectAdapter(fileLogApi,dbLogApi);

        //先测试从文件操作适配到第二版
        //虽然调用的是第二版的接口，其实是文件操作在实现
        dbLogApi2.createLog(lm1);
        List<LogModel> allLog = dbLogApi2.getAllLog();
        System.out.println("allLog="+allLog);

        //再测试从数据库存储适配成第一版的接口
        //也就是调用第一版的接口，其实是数据实现
        fileLogApi2.writeLogFile(list);
        fileLogApi2.readLogFile();
    }
}
```

运行一下，看看结果，体会一下双向适配器。

> 注意
>
> **事实上，使用适配器有一个潜在的问题，就是被适配的对象不再兼容 Adaptee 的接口，因为适配器只是实现了 Target 的接口。这导致并不是所有 Adaptee 对象可以被使用的地方都能使用适配器。**

而双向适配器就解决了这样的问题，双向适配器同时实现了 Target 和 Adaptee 的接口，使得双向适配器可以在 Target 或 Adaptee 被使用的地方使用，以提供对所有客户的透明性。尤其在两个不同的客户需要用不同的方式查看同一个对象时，适合使用双向适配器。

4.3.4 对象适配器和类适配器

在标准的适配器模式里面，根据适配器的实现方式，把适配器分成了两种，一种是对象适配器，另一种是类适配器。

对象适配器的实现：依赖于对象组合。就如同前面的实现示例，都是采用对象组合的方式，也就是对象适配器实现的方式。

类适配器的实现：采用多重继承对一个接口与另一个接口进行匹配。由于 Java 不支持多重继承，所以到目前为止还没有涉及。

1. 类适配器

前面已经学习过对象适配器了，下面简单地介绍一下类适配器。首先来看看类适配器的结构，如图 4.13 所示。

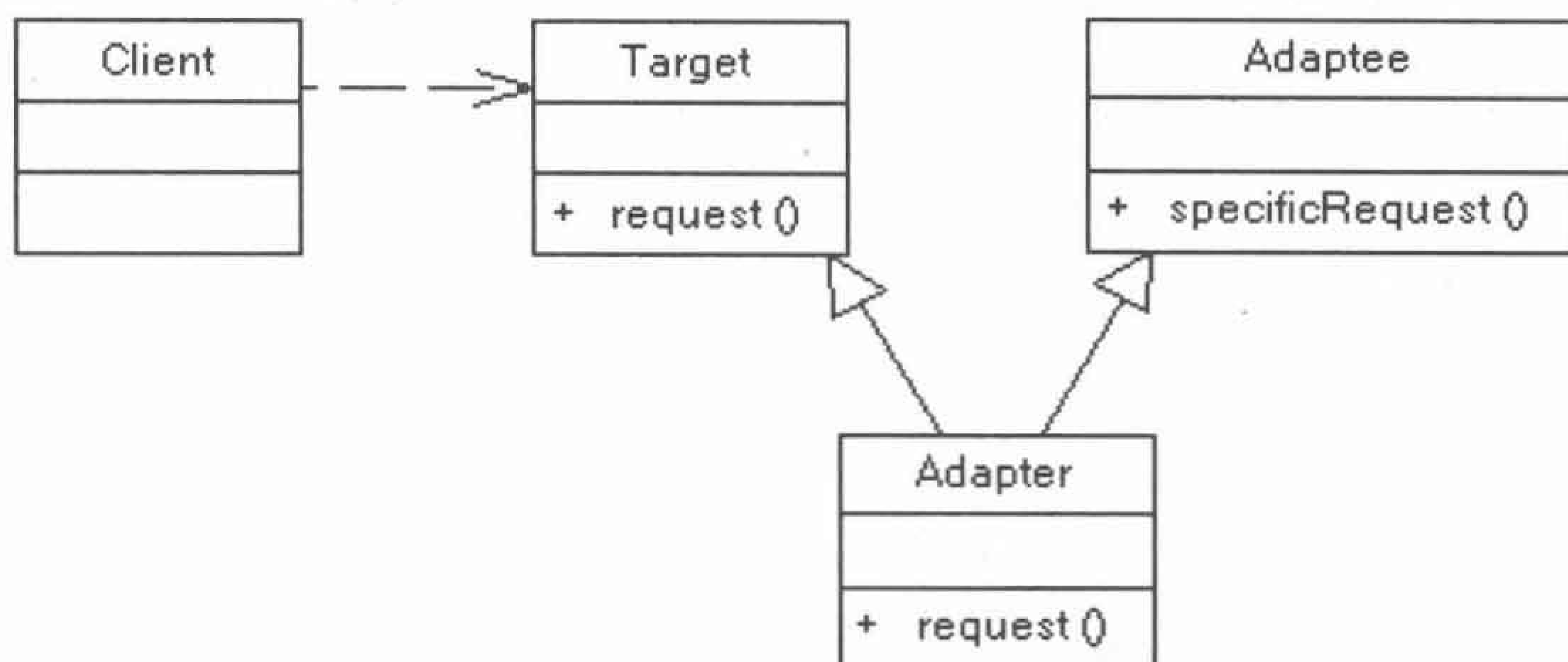


图 4.13　类适配器的结构示意图

从结构图上可以看出，类适配器是通过继承来实现接口适配的，标准的设计模式中，类适配器是同时继承 Target 和 Adaptee 的，也就是一个多重继承，这在 Java 里面是不被支持的，也就是说 Java 中是不能实现标准的类适配器的。

但是 Java 中有一种变通的方式，也能够使用继承来实现接口的适配，那就是让适配器去实现 Target 的接口，然后继承 Adaptee 的实现，虽然不是十分标准，但是意思差不多。下面就来看个小示例。

2. Java 中类似实现类适配器的例子

还是来实现前面的那个示例，就是让文件存储日志的实现，能够经过适配，满足第二版日志操作接口的要求。

基本的实现方式是：写一个适配器类，让适配器类去继承文件存储日志的实现，然后让适配器类去实现第二版日志操作接口的要求。

这样实现的示例整体结构如图 4.14 所示。

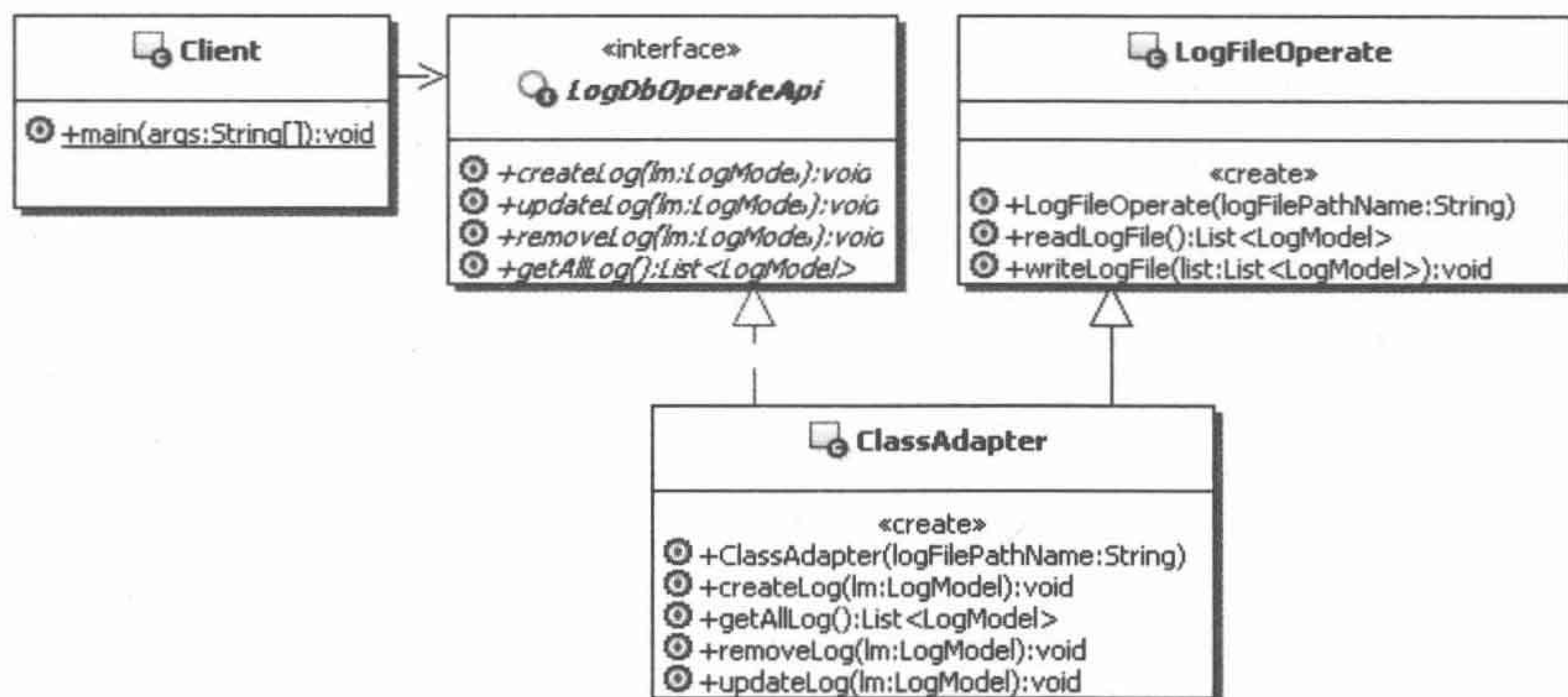


图 4.14　类似类适配器示例的结构示意图

在实现中，主要是适配器的实现与以前不一样，与对象适配器实现同样的功能相比，类适配器在实现上有如下改变。

- 需要继承 LogFileOperate 的实现，然后再实现 LogDbOperateApi 接口。
- 需要按照继承 LogFileOperate 的要求，提供传入文件路径和名称的构造方法。
- 不再需要持有 LogFileOperate 的对象了，因为适配器本身就是 LogFileOperate 对象的子类了。
- 以前调用被适配对象的方法的地方，全部修改成调用自己的方法。

真正功能的实现，类适配器和对象适配器两种方式都差不多。示例代码如下：

```
/**
 * 类适配器对象
 */
public class ClassAdapter
            extends LogFileOperate implements LogDbOperateApi{
    public ClassAdapter(String logFilePathName) {
        super(logFilePathName);
    }

    public void createLog(LogModel lm) {
        //1: 先读取文件的内容
        List<LogModel> list = this.readLogFile();
        //2: 加入新的日志对象
        list.add(lm);
        //3: 重新写入文件
        this.writeLogFile(list);
    }
    public List<LogModel> getAllLog() {
        return this.readLogFile();
    }
    public void removeLog(LogModel lm) {
        //1: 先读取文件的内容
        List<LogModel> list = this.readLogFile();
        //2: 删除相应的日志对象
        list.remove(lm);
        //3: 重新写入文件
        this.writeLogFile(list);
    }
    public void updateLog(LogModel lm) {
        //1: 先读取文件的内容
        List<LogModel> list = this.readLogFile();
```

```
        //2：修改相应的日志对象
        for(int i=0;i<list.size();i++){
            if(list.get(i).getLogId().equals(lm.getLogId())){
                list.set(i, lm);
                break;
            }
        }
        //3：重新写入文件
        this.writeLogFile(list);
    }
}
```

自己写个客户端去测试看看，体会一下。

3. 类适配器和对象适配器的权衡

- 从实现上：类适配器使用对象继承的方式，是静态的定义方式；而对象适配器使用对象组合的方式，是动态组合的方式
- 对于类适配器，由于适配器直接继承了 Adaptee，使得适配器不能和 Adaptee 的子类一起工作，因为继承是静态的关系，当适配器继承了 Adaptee 后，就不可能再去处理 Adaptee 的子类了。

  对于对象适配器，允许一个 Adapter 和多个 Adaptee，包括 Adaptee 和它所有的子类一起工作。因为对象适配器采用的是对象组合的关系，只要对象类型正确，是不是子类都无所谓。
- 对于类适配器，适配器可以重定义 Adaptee 的部分行为，相当于子类覆盖父类的部分实现方法。

  对于对象适配器，要重定义 Adaptee 的行为比较困难，这种情况下，需要定义 Adaptee 的子类来实现重定义，然后让适配器组合子类。
- 对于类适配器，仅仅引入了一个对象，并不需要额外的引用来间接得到 Adaptee。对于对象适配器，需要额外的引用来间接得到 Adaptee。

  在 Java 开发中，建议大家尽量使用对象适配器的实现方式。当然，具体问题具体分析，根据需要来选用实现方式，最合适的才是最好的。

4.3.5　适配器模式的优缺点

适配器模式有如下优点。

- 更好的复用性

  如果功能是已经有了的，只是接口不兼容，那么通过适配器模式就可以让这些功能得到更好的复用。
- 更好的可扩展性

  在实现适配器功能的时候，可以调用自己开发的功能，从而自然地扩展系统的

功能。

适配器模式有如下缺点。

- 过多地使用适配器，**会让系统非常零乱，不容易整体进行把握**

  比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口来实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

4.3.6 思考适配器模式

1. 适配器模式的本质

> **适配器模式的本质是：转换匹配，复用功能。**

适配器通过转换调用已有的实现，从而能把已有的实现匹配成需要的接口，使之能满足客户端的需要。也就是说转换匹配是手段，而复用已有的功能才是目的。

在进行转换匹配的过程中，适配器还可以在转换调用的前后实现一些功能处理，也就是实现智能的适配。

2. 何时选用适配器模式

建议在以下情况中选用适配器模式。

- 如果你想要使用一个已经存在的类，但是它的接口不符合你的需求，这种情况可以使用适配器模式，来把已有的实现转换成你需要的接口。
- 如果你想创建一个可以复用的类，这个类可能和一些不兼容的类一起工作，这种情况可以使用适配器模式，到时候需要什么就适配什么。
- 如果你想使用一些已经存在的子类，但是不可能对每一个子类都进行适配，这种情况可以选用对象适配器，直接适配这些子类的父类就可以了。

4.3.7 相关模式

- 适配器模式与桥接模式

  其实这两个模式除了结构略为相似外，功能上完全不同。

  适配器模式是把两个或者多个接口的功能进行转换匹配；而桥接模式是让接口和实现部分相分离，以便它们可以相对独立地变化。

- 适配器模式与装饰模式

  从某种意义上讲，适配器模式能模拟实现简单的装饰模式的功能，也就是为已有功能增添功能。比如我们在适配器里面这么写：

```
public void adapterMethod(){
    System.out.println("在调用 Adaptee 的方法之前完成一定的工作");
```

```
    //调用 Adaptee 的相关方法
    adaptee.method();
    System.out.println("在调用 Adaptee 的方法之后完成一定的工作");
}
```

如上的写法，就相当于在调用 Adaptee 的被适配方法前后添加了新的功能，这样适配过后，客户端得到的功能就不单纯是 Adaptee 的被适配方法的功能了。看看是不是类似装饰模式的功能呢？

> 注意
>
> **注意，仅仅是类似，造成这种类似的原因是：两种设计模式在实现上都是使用的对象组合，都可以在转调组合对象的功能前后进行一些附加的处理，因此有这么一个相似性。它们的目的和本质都是不一样的。**

两个模式有一个很大的不同：一般适配器适配过后是需要改变接口的，如果不改接口就没有必要适配了；而装饰模式是不改变接口的，无论多少层装饰都是一个接口。因此装饰模式可以很容易地支持递归组合，而适配器就做不到，每次的接口不同，无法递归。

- 适配器模式和代理模式

  适配器模式可以和代理模式组合使用。在实现适配器的时候，可以通过代理来调用 Adaptee，这样可以获得更大的灵活性。

- 适配器模式和抽象工厂模式

  在适配器实现的时候，通常需要得到被适配的对象。如果被适配的是一个接口，那么就可以结合一些可以创造对象实例的设计模式，来得到被适配的对象示例，比如抽象工厂模式、单例模式、工厂方法模式等。

读书笔记

第 5 章　单例模式（Singleton）

5.1 场景问题

5.1.1 读取配置文件的内容

考虑这样一个应用，读取配置文件的内容。

很多应用项目，都有与应用相关的配置文件，这些配置文件很多是由项目开发人员自定义的，在里面定义一些应用需要的参数数据。当然在实际的项目中，这种配置文件多采用 xml 格式，也有采用 properties 格式的，毕竟使用 Java 来读取 properties 格式的配置文件比较简单。

现在要读取配置文件的内容，该如何实现呢？

5.1.2 不用模式的解决方案

有些朋友会想，要读取配置文件的内容，这也不是个困难的事情，直接读取文件的内容，然后把文件内容存放在相应的数据对象里面就可以了。真的这么简单吗？先实现看看吧。

为了示例简单，假设系统采用的是 properties 格式的配置文件。

（1）直接使用 Java 来读取配置文件的示例代码如下：

```
/**
 * 读取应用配置文件
 */
public class AppConfig {
    /**
     * 用来存放配置文件中参数 A 的值
     */
    private String parameterA;
    /**
     * 用来存放配置文件中参数 B 的值
     */
    private String parameterB;

    public String getParameterA() {
        return parameterA;
    }
    public String getParameterB() {
        return parameterB;
    }
    /**
```

注意：只有访问参数的方法，没有设置参数的方法

```
     * 构造方法
     */
    public AppConfig(){
        //调用读取配置文件的方法
        readConfig();
    }
    /**
     * 读取配置文件，把配置文件中的内容读出来设置到属性上
     */
    private void readConfig(){
        Properties p = new Properties();
        InputStream in = null;
        try {
            in = AppConfig.class.getResourceAsStream(
                                    "AppConfig.properties");
            p.load(in);
            //把配置文件中的内容读出来设置到属性上
            this.parameterA = p.getProperty("paramA");
            this.parameterB = p.getProperty("paramB");
        } catch (IOException e) {
            System.out.println("装载配置文件出错了，具体堆栈信息如下：");
            e.printStackTrace();
        }finally{
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

（2）应用的配置文件，名字是 AppConfig.properties，放在 AppConfig 相同的包里面。简单示例如下：

```
paramA=a
paramB=b
```

（3）写个客户端来测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建读取应用配置的对象
```

```
        AppConfig config = new AppConfig();

        String paramA = config.getParameterA();
        String paramB = config.getParameterB();

        System.out.println("paramA="+paramA+",paramB="+paramB);
    }
}
```

运行结果如下：

```
paramA=a,paramB=b
```

5.1.3 有何问题

上面的实现很简单，很容易的就实现要求的功能。仔细想想，有没有什么问题呢？

看看客户端使用这个类的地方，是通过 new 一个 AppConfig 的实例来得到一个操作配置文件内容的对象。如果在系统运行中，有很多地方都需要使用配置文件的内容，也就是说很多地方都需要创建 AppConfig 对象的实例。

换句话说，在系统运行期间，系统中会存在很多个 AppConfig 的实例对象，这有什么问题吗？

当然有问题了，试想一下，每一个 AppConfig 实例对象里面都封装着配置文件的内容，系统中有多个 AppConfig 实例对象，也就是说系统中会同时存在多份配置文件的内容，这样会严重浪费内存资源。如果配置文件内容较少，问题还小一点，如果配置文件内容本来就多的话，对于系统资源的浪费问题就大了。事实上，对于 AppConfig 这种类，在运行期间，只需要一个实例对象就是够了。

把上面的描述进一步抽象一下，问题就出来了：在一个系统运行期间，某个类只需要一个类实例就可以了，那么应该怎样实现呢？

5.2 解决方案

5.2.1 使用单例模式来解决问题

用来解决上述问题的一个合理的解决方案就是单例模式（Singleton）。那么什么是单例模式呢？

1. 单例模式的定义

> **保证一个类仅有一个实例，并提供一个访问它的全局访问点。**

2. 应用单例模式来解决问题的思路

仔细分析上面的问题，现在一个类能够被创建多个实例，问题的根源在于类的构造方法是公开的，也就是可以让类的外部来通过构造方法创建多个实例。换句话说，只要类的构造方法能让类的外部访问，就没有办法去控制外部来创建这个类的实例个数。

要想控制一个类只被创建一个实例，那么首要的问题就是要把创建实例的权限收回来，让类自身来负责自己类实例的创建工作，然后由这个类来提供外部可以访问这个类实例的方法，这就是单例模式的实现方式。

5.2.2　单例模式的结构和说明

单例模式的结构如图 5.1 所示。

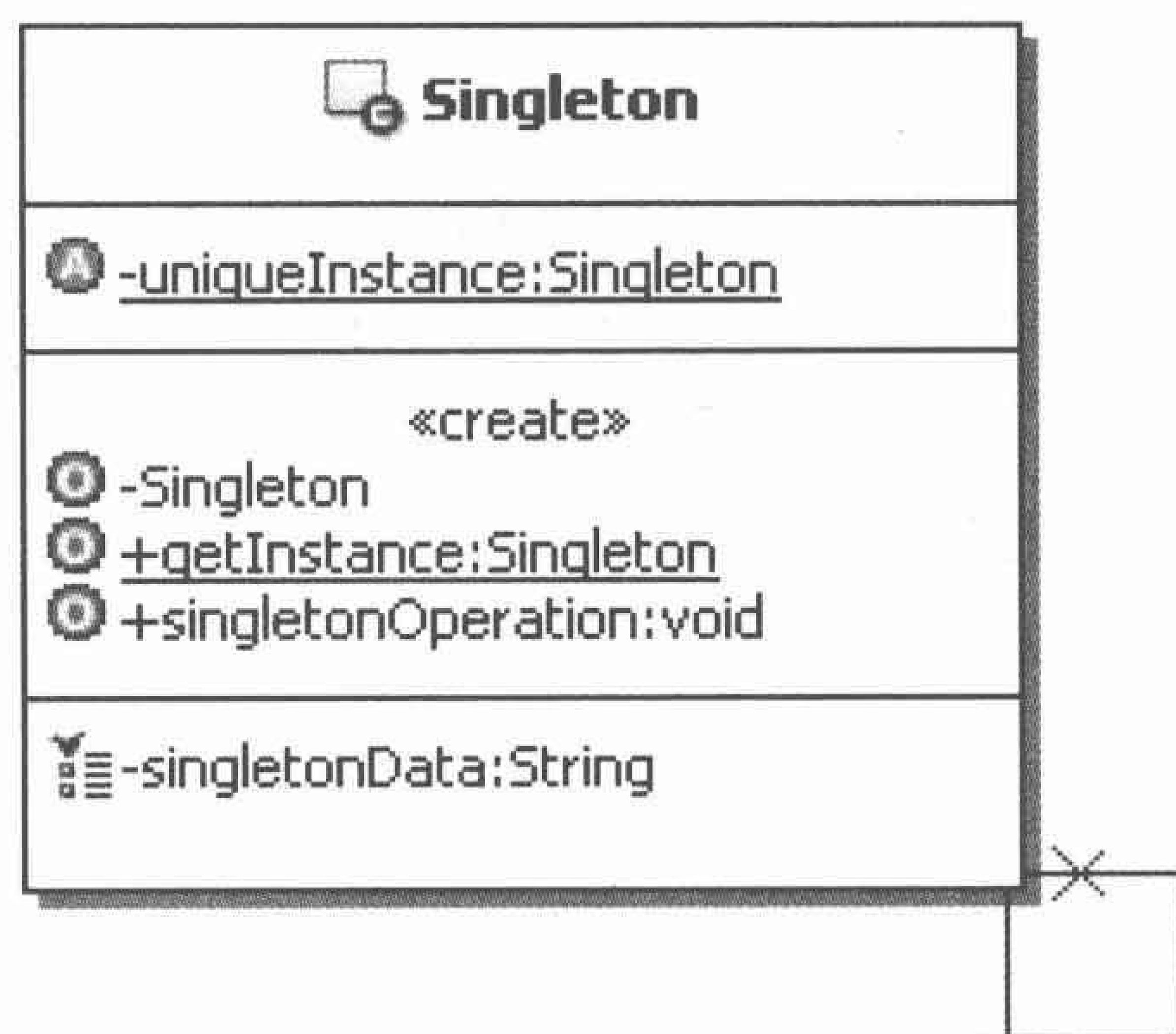


图 5.1　单例模式结构图

Singleton：负责创建 Singleton 类自己的唯一实例，并提供一个 getInstance 的方法，让外部来访问这个类的唯一实例。

5.2.3　单例模式示例代码

在 Java 中，单例模式的实现又分为两种，一种称为懒汉式，一种称为饿汉式，其实就是在具体创建对象实例的处理上，有不同的实现方式。下面分别来看看这两种实现方式的代码示例。为何这么写，具体在后面再来讲述。

（1）懒汉式实现示例代码如下：

```
/**
 * 懒汉式单例实现的示例
 */
public class Singleton {
    /**
     * 定义一个变量来存储创建好的类实例
     */
    private static Singleton uniqueInstance = null;
```

```
    /**
     * 私有化构造方法，可以在内部控制创建实例的数目
     */
    private Singleton(){
        //
    }
    /**
     * 定义一个方法来为客户端提供类实例
     * @return 一个 Singleton 的实例
     */
    public static synchronized Singleton getInstance(){
        //判断存储实例的变量是否有值
        if(uniqueInstance == null){
            //如果没有，就创建一个类实例，并把值赋值给存储类实例的变量
            uniqueInstance = new Singleton();
        }
        //如果有值，那就直接使用
        return uniqueInstance;
    }
    /**
     * 示意方法，单例可以有自己的操作
     */
    public void singletonOperation(){
        //功能处理
    }
    /**
     * 示意属性，单例可以有自己的属性
     */
    private String singletonData;
    /**
     * 示意方法，让外部通过这些方法来访问属性的值
     * @return 属性的值
     */
    public String getSingletonData(){
        return singletonData;
    }
}
```

（2）饿汉式实现，示例代码如下：

```
/**
 * 饿汉式单例实现的示例
```

```
 */
public class Singleton {
    /**
     * 定义一个变量来存储创建好的类实例，直接在这里创建类实例，只能创建一次
     */
    private static Singleton uniqueInstance = new Singleton();
    /**
     * 私有化构造方法，可以在内部控制创建实例的数目
     */
    private Singleton(){
        //
    }
    /**
     * 定义一个方法来为客户端提供类实例
     * @return 一个 Singleton 的实例
     */
    public static Singleton getInstance(){
        //直接使用已经创建好的实例
        return uniqueInstance;
    }

    /**
     * 示意方法，单例可以有自己的操作
     */
    public void singletonOperation(){
        //功能处理
    }
    /**
     * 示意属性，单例可以有自己的属性
     */
    private String singletonData;
    /**
     * 示意方法，让外部通过这些方法来访问属性的值
     * @return 属性的值
     */
    public String getSingletonData(){
        return singletonData;
    }
}
```

关于饿汉式、懒汉式的名称说明：

饿汉式、懒汉式其实是一种比较形象的称谓。

所谓饿汉式，既然饿，那么在创建对象实例的时候就比较着急，饿了嘛，于是就在装载类的时候就创建对象实例，写法如下：

```
private static Singleton uniqueInstance = new Singleton();
```

所谓懒汉式，既然是懒，那么在创建对象实例的时候就不着急，会一直等到马上要使用对象实例的时候才会创建，懒人嘛，总是推托不开的时候才去真正执行工作，因此在装载对象的时候不创建对象实例，写法如下：

```
private static Singleton uniqueInstance = null;
```

延伸 **而是等到第一次使用的时候，才去创建实例，也就是在 getInstance 方法里面去判断和创建**

5.2.4 使用单例模式重写示例

由于单例模式有两种实现方式，这里选择一种来实现就可以了，我们选择饿汉式的实现方式来重写示例吧。

采用饿汉式的实现方式来重写实例的示例代码如下：

```
/**
 * 读取应用配置文件，单例实现
 */
public class AppConfig {
    /**
     * 定义一个变量来存储创建好的类实例，直接在这里创建类实例，只能创建一次
     */
    private static AppConfig instance = new AppConfig();
    /**
     * 定义一个方法来为客户端提供 AppConfig 类的实例
     * @return 一个 AppConfig 的实例
     */
    public static AppConfig getInstance(){
        return instance;
    }

    /**
     * 用来存放配置文件中参数 A 的值
     */
    private String parameterA;
    /**
```

```
     * 用来存放配置文件中参数 B 的值
     */
    private String parameterB;
    public String getParameterA() {
        return parameterA;
    }
    public String getParameterB() {
        return parameterB;
    }
    /**
     * 私有化构造方法
     */
    private AppConfig(){
        //调用读取配置文件的方法
        readConfig();
    }
    /**
     * 读取配置文件，把配置文件中的内容读出来设置到属性上
     */
    private void readConfig(){
        Properties p = new Properties();
        InputStream in = null;
        try {
            in = AppConfig.class.getResourceAsStream(
                                        "AppConfig.properties");
            p.load(in);
            //把配置文件中的内容读出来设置到属性上
            this.parameterA = p.getProperty("paramA");
            this.parameterB = p.getProperty("paramB");
        } catch (IOException e) {
            System.out.println("装载配置文件出错了，具体堆栈信息如下：");
            e.printStackTrace();
        }finally{
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}
```

当然，测试的客户端也需要相应地变化。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建读取应用配置的对象
        AppConfig config = AppConfig.getInstance();

        String paramA = config.getParameterA();
        String paramB = config.getParameterB();

        System.out.println("paramA="+paramA+",paramB="+paramB);
    }
}
```

测试看看，是否能满足要求。

5.3 模式讲解

5.3.1 认识单例模式

1. 单例模式的功能

单例模式是用来保证这个类在运行期间只会被创建一个类实例，另外，单例模式还提供了一个全局唯一访问这个类实例的访问点，就是 getInstance 方法。不管采用懒汉式还是饿汉式的实现方式，这个全局访问点是一样的。

对于单例模式而言，不管采用何种实现方式，它都是只关心类实例的创建问题，并不关心具体的业务功能。

2. 单例模式的范围

也就是在多大范围内是单例呢？

观察上面的实现可以知道，目前 Java 里面实现的单例是一个虚拟机的范围。因为装载类的功能是虚拟机的，所以一个虚拟机在通过自己的 ClassLoader 装载饿汉式实现单例类的时候就会创建一个类的实例。

说明	**就把单例模式中的 5.3.1 的认识单例模式里面的第（2）这个小项的内容，替换成下面的内容即可。**

这就意味着如果一个虚拟机里面有很多个 ClassLoader，而且这些 ClassLoader 都装载某个类的话，就算这个类是单例，它也会产生很多个实例。当然，如果一个机器上有多个虚拟机，那么每个虚拟机里面都应该至少有一个这个类的实例，也就是说整个机器

上就有很多个实例，更不会是单例了。

3. 单例模式的命名

注意	**另外请注意一点，这里讨论的单例模式并不适用于集群环境，对于集群环境下的单例这里不去讨论，它不属于这里的内容范围。**

一般建议单例模式的方法命名为 getInstance()，这个方法的返回类型肯定是单例类的类型了。getInstance()方法可以有参数，这些参数可能是创建类实例所需要的参数，当然，大多数情况下是不需要的。

单例模式的名称有单例、单件、单体等，只是翻译的不同，都是指的同一个模式。

5.3.2　懒汉式和饿汉式实现

前面提到了单例模式有两种典型的解决方案，一种叫懒汉式，另一种叫饿汉式，这两种方式究竟是如何实现的，下面分别来看看。为了看得更清晰一点，只是实现基本的单例控制部分，不再提供示例的属性和方法了；而且暂时也不去考虑线程安全的问题，这个问题在后面将会重点分析。

1. 第一种方案——懒汉式

1）私有化构造方法

要想在运行期间控制某一个类的实例只有一个，首要的任务就是要控制创建实例的地方，也就是不能随随便便就可以创建类实例，否则就无法控制所创建的实例个数了。现在是让使用类的地方来创建类实例，也就是在类外部来创建类实例。

那么怎样才能让类的外部不能创建一个类的实例呢？很简单，私有化构造方法就可以了。示例代码如下：

```
private Singleton(){
}
```

2）提供获取实例的方法

构造方法被私有化了，外部使用这个类的地方不干了，外部创建不了类实例就没有办法调用这个对象的方法，就实现不了功能调用。这可不行，经过思考，单例模式决定让这个类提供一个方法来返回类的实例，方便外面使用。示例代码如下：

```
public Singleton getInstance(){
}
```

3）把获取实例的方法变成静态的

又有新的问题了，获取对象实例的这个方法是一个实例方法，也就是说客户端要想调用这个方法，需要先得到类实例，然后才可以调用。可是这个方法就是为了得到类实例，这样一来不就形成一个死循环了吗？这也是典型的“先有鸡还是先有蛋的问题”。

解决方法也很简单，在方法上加上 static，这样就可以直接通过类来调用这个方法，而不需要先得到类实例。示例代码如下：

```
public static Singleton getInstance(){
}
```

4）定义存储实例的属性

方法定义好了，那么方法内部如何实现呢？如果直接创建实例并返回，这样行不行呢？示例代码如下：

```
public static Singleton getInstance(){
   return new Singleton();
}
```

当然不行了，如果每次客户端访问都这样直接 new 一个实例，那肯定会有多个实例，根本实现不了单例的功能。

怎么办呢？单例模式想到了一个办法，那就是用一个属性来记录自己创建好的类实例。当第一次创建后，就把这个实例保存下来，以后就可以复用这个实例，而不是重复创建对象实例了。示例代码如下：

```
private Singleton instance = null;
```

5）把这个属性也定义成静态的

这个属性变量应该在什么地方用呢？肯定是第一次创建类实例的地方，也就是在前面那个返回对象实例的静态方法里面使用。

由于要在一个静态方法里面使用，所以这个属性被迫成为一个类变量，要强制加上 static，也就是说，这里并没有使用 static 的特性。示例代码如下：

```
private static Singleton instance = null;
```

6）实现控制实例的创建

现在应该到 getInstance 方法里面实现控制实例的创建了。控制的方式很简单，只要先判断一下是否已经创建过实例就可以了。如何判断？那就看存放实例的属性是否有值，如果有值，说明已经创建过了，如果没有值，则应该创建一个。示例代码如下：

```
public static Singleton getInstance(){
    //先判断 instance 是否有值
    if(instance == null){
        //如果没有值，说明还没有创建过实例，那就创建一个
        //并把这个实例设置给 instance
        instance = new Singleton ();
    }
    //如果有值，或者是创建了值，那就直接使用
    return instance;
}
```

7）完整的实现

至此，成功解决了在运行期间，控制某个类只被创建一个实例的要求。完整的代码如下。为了大家好理解，用注释标示了代码的先后顺序。

```
public class Singleton {
    //4：定义一个变量来存储创建好的类实例
    //5：因为这个变量要在静态方法中使用，所以需要加上 static 修饰
    private static Singleton instance = null;
    //1：私有化构造方法，好在内部控制创建实例的数目
    private Singleton(){
    }
    //2：定义一个方法来为客户端提供类实例
    //3：这个方法需要定义成类方法，也就是要加 static
    public static Singleton getInstance(){
        //6：判断存储实例的变量是否有值
        if(instance == null){
            //6.1：如果没有，就创建一个类实例，并把值赋给存储类实例的变量
            instance = new Singleton();
        }
        //6.2：如果有值，那就直接使用
        return instance;
    }
}
```

2. 第二种方案——饿汉式

这种方案和第一种方案相比，前面的私有化构造方法，提供静态的 getInstance 方法来返回实例等步骤都一样。差别在于如何实现 getInstance 方法，在这个地方，单例模式还想到了另外一种方法来实现 getInstance 方法。

不就是要控制只创造一个实例吗？那么有没有什么现成的解决办法呢？很快，单例模式回忆起了 Java 中 static 的特性。

- static 变量在类装载的时候进行初始化。
- 多个实例的 static 变量会共享同一块内存区域。

这就意味着，在 Java 中，static 变量只会被初始化一次，就是在类装载的时候，而且多个实例都会共享这个内存空间，这不就是单例模式要实现的功能吗？真是得来全不费功夫啊。根据这些知识，写出了第二种解决方案的代码。

```
public class Singleton {
    //4：定义一个静态变量来存储创建好的类实例
    //直接在这里创建类实例，只能创建一次
    private static Singleton instance = new Singleton();
    //1：私有化构造方法，可以在内部控制创建实例的数目
    private Singleton(){
    }
    //2：定义一个方法来为客户端提供类实例
    //3：这个方法需要定义成类方法，也就是要加 static
```

注意在这里就创建类实例了

```
    public static Singleton getInstance(){
        //5: 直接使用已经创建好的实例
        return instance;
    }
}
```

这个方法里面就不需要控制代码了

提示 **注意一下，这个方案用到了 static 的特性，而第一个方案却没有用到，因此两个方案的步骤会有一些不同。在第一个方案里面，强制加上 static 也是算作一步的，而在这个方案里面，是主动加上 static，就不能单独算作一步了。**

所以在查看上面两种方案代码的时候，仔细看看编号。顺着编号的顺序看，可以体会出两种方案的不一样。

不管是采用哪一种方式，在运行期间，都只会生成一个实例，而访问这些类的一个全局访问点，就是那个静态的 getInstance 方法。

3. 单例模式的调用顺序示意图

由于单例模式有两种实现方式，所以它的调用顺序也分成两种。

先来看懒汉式的调用顺序，如图 5.2 所示。

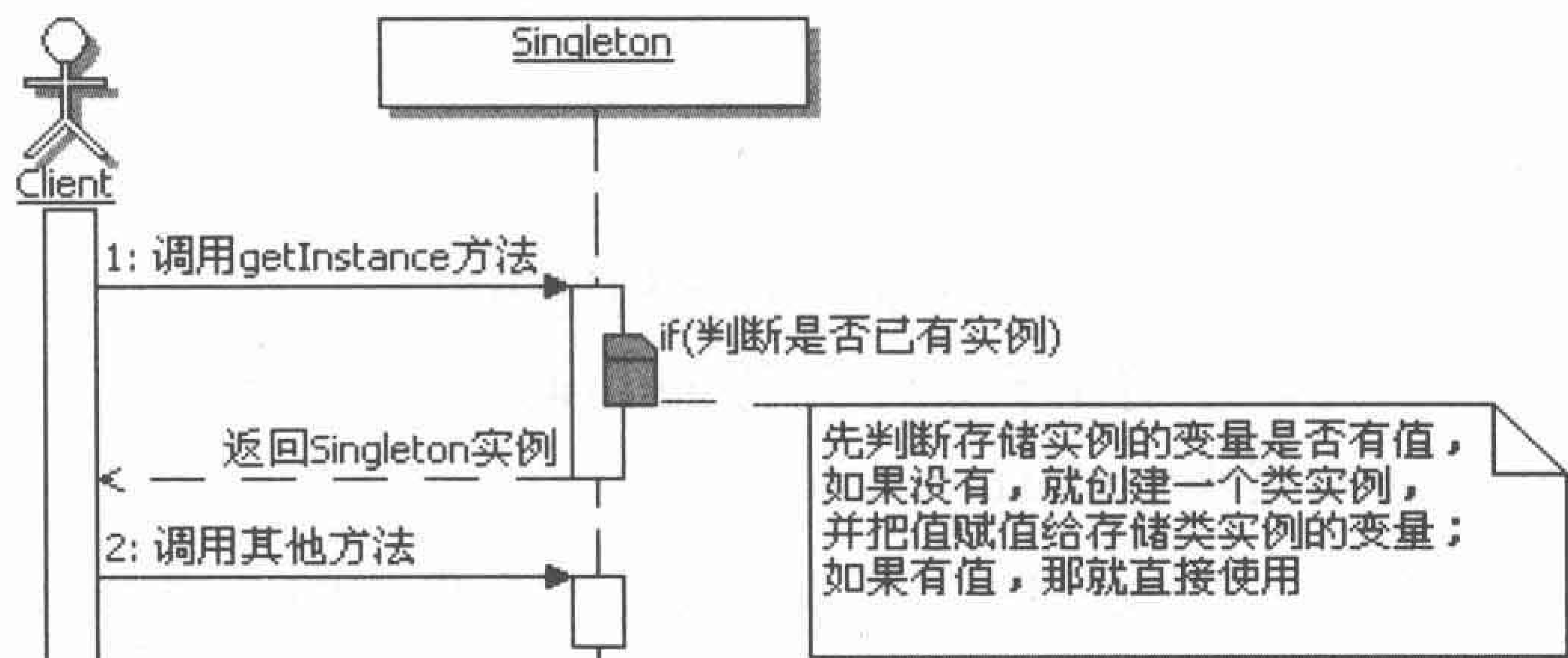


图 5.2　懒汉式调用顺序示意图

饿汉式的调用顺序如图 5.3 所示。

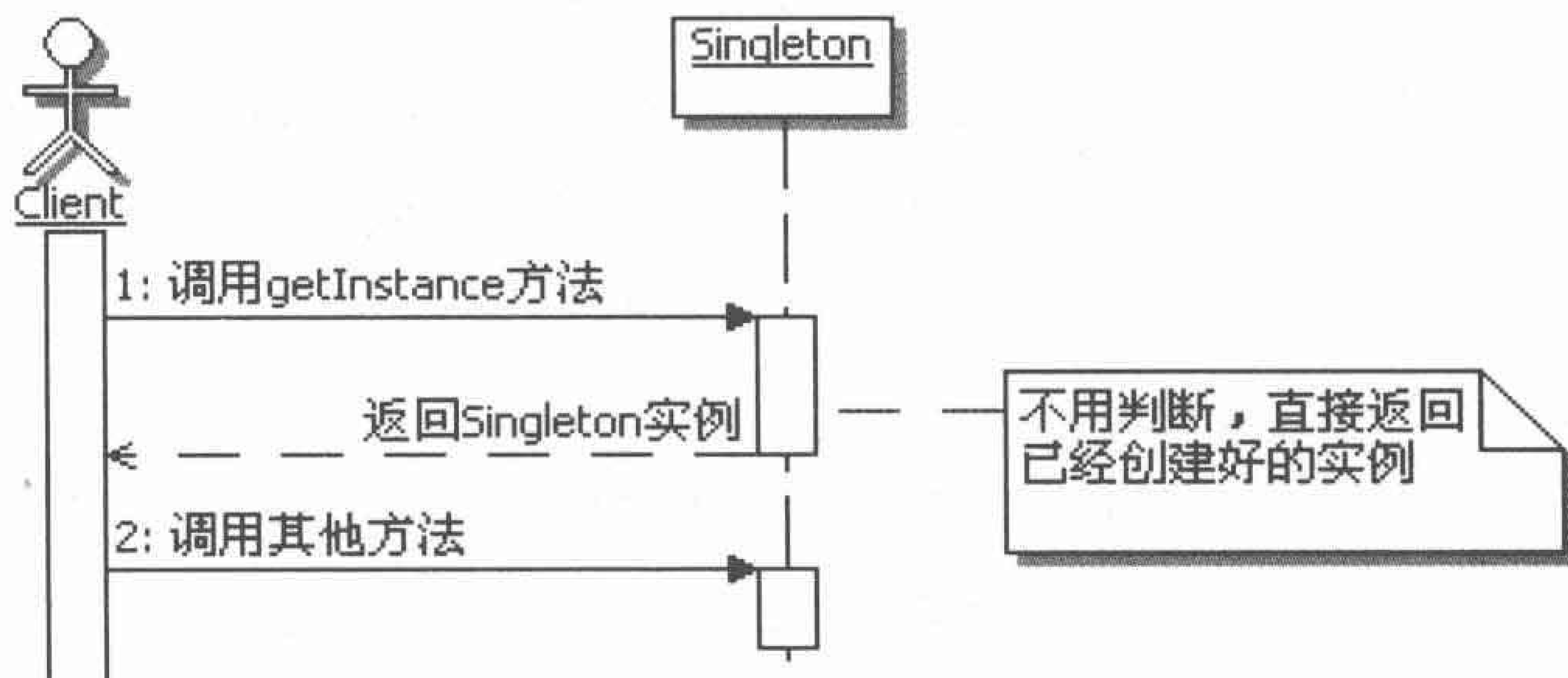


图 5.3　饿汉式调用顺序示意图

5.3.3　延迟加载的思想

单例模式的懒汉式实现方式体现了延迟加载的思想。什么是延迟加载呢？

通俗点说，延迟加载就是一开始不要加载资源或者数据，一直等，等到马上就要使用这个资源或者数据了，躲不过去了才加载，所以也称 Lazy Load，不是懒惰啊，是“延迟加载”，这在实际开发中是一种很常见的思想，尽可能地节约资源。

体现在什么地方呢？请看如下代码：

```
public static Singleton getInstance(){

    if(instance == null){
        instance = new Singleton();
    }
    return instance;

}
```

这里就体现了延迟加载，马上就要使用这个实例了，还不知道有没有呢，所以判断一下，如果没有，没办法了，赶紧创建一个吧。

5.3.4　缓存的思想

单例模式的懒汉式实现还体现了缓存的思想，缓存也是实际开发中常见的功能。

简单讲就是，当某些资源或者数据被频繁地使用，而这些资源或数据存储在系统外部，比如数据库、硬盘文件等，那么每次操作这些数据的时候都得从数据库或者硬盘上去获取，速度会很慢，将造成性能问题。

一个简单的解决方法就是：把这些数据缓存到内存里面，每次操作的时候，先到内存里面找，看有没有这些数据，如果有，就直接使用，如果没有就获取它，并设置到缓存中，下一次访问的时候就可以直接从内存中获取了，从而节省大量的时间。当然，**缓存是一种典型的空间换时间的方案**。

缓存在单例模式的实现中是怎样体现的呢？

```
public class Singleton {
    private static Singleton instance = null;
    private Singleton(){

    }
    public static Singleton getInstance(){
        //判断存储实例的变量是否有值
        if(instance == null){
            //如果没有，就创建一个类实例，并把值赋给存储类实例的变量
            instance = new Singleton();
        }
        //如果有值，那就直接使用
        return instance;
```

这个属性就是用来缓存实例的

缓存的实现

```
    }
}
```

5.3.5 Java 中缓存的基本实现

下面来看看在 Java 开发中缓存的基本实现，在 Java 开发中最常见的一种实现缓存的方式就是使用 Map，基本步骤如下。

（1）先到缓存里面查找，看看是否存在需要使用的数据。

（2）如果没有找到，那么就创建一个满足要求的数据，然后把这个数据设置到缓存中，以备下次使用。如果找到了相应的数据，或者是创建了相应的数据，那就直接使用这个数据。

还是看看示例吧。示例代码如下：

```
/**
 * Java 中缓存的基本实现示例
 */
public class JavaCache {
    /**
     * 缓存数据的容器，定义成 Map 是方便访问，直接根据 key 就可以获取 Value 了
     * key 选用 String 是为了简单，方便演示
     */
    private Map<String,Object> map = new HashMap<String,Object>();
    /**
     * 从缓存中获取值
     * @param key 设置时候的 key 值
     * @return key 对应的 Value 值
     */
    public Object getValue(String key){
        //先从缓存里面取值
        Object obj = map.get(key);
        //判断缓存里面是否有值
        if(obj == null){
            //如果没有，那么就去获取相应的数据，比如读取数据库或者文件
            //这里只是演示，所以直接写个假的值
            obj = key+",value";
            //把获取的值设置回到缓存里面
            map.put(key, obj);
        }
        //如果有值了，就直接返回使用
        return obj;
    }
```

```
}
```

这里只是缓存的基本实现，还有很多功能都没有考虑，比如缓存的清除，缓存的同步等。当然，Java 的缓存还有很多实现方式，也是非常复杂的，现在有很多专业的缓存框架。更多缓存的知识，这里就不再讨论了。

5.3.6　利用缓存来实现单例模式

应用 Java 缓存的知识，可以变相实现 Singleton 模式，也算是一个模拟实现吧。每次都先从缓存中取值。只要创建一次对象实例后，就设置了缓存的值，那么下次就不用再创建了。

虽然不是很标准的做法，但是同样可以实现单例模式的功能。为了简单，先不去考虑多线程的问题，示例代码如下：

```
/**
 * 使用缓存来模拟实现单例
 */
public class Singleton {
    /**
     * 定义一个默认的 key 值，用来标识在缓存中的存放
     */
    private final static String DEFAULT_KEY = "One";
    /**
     * 缓存实例的容器
     */
    private static Map<String,Singleton> map =
                                new HashMap<String,Singleton>();
    /**
     * 私有化构造方法
     */
    private Singleton(){
        //
    }
    public static Singleton getInstance(){
        //先从缓存中获取
        Singleton instance = (Singleton)map.get(DEFAULT_KEY);
        //如果没有，就新建一个，然后设置回缓存中
        if(instance==null){
            instance = new Singleton();
            map.put(DEFAULT_KEY, instance);
        }
```

```
    }
}
```

是不是也能实现单例所要求的功能呢？前面讲过，实现模式的方式有很多种，并不是只有模式的参考实现所实现的方式，上面这种也能实现单例所要求的功能，只不过实现比较麻烦，不是太好而已，但在后面扩展单例模式的时候会有用。

另外，前面也讲过，模式是经验的积累，模式的参考实现并不一定是最优的，对于单例模式，后面将会给大家一些更好的实现方式。

5.3.7 单例模式的优缺点

1. 时间和空间

比较上面两种写法：**懒汉式是典型的时间换空间**，也就是每次获取实例都会进行判断，看是否需要创建实例，浪费判断的时间。当然，如果一直没有人使用的话，那就不会创建实例，则节约内存空间。

饿汉式是典型的空间换时间，当类装载的时候就会创建类实例，不管你用不用，先创建出来，然后每次调用的时候，就不需要再判断了，节省了运行时间。

2. 线程安全

（1）从线程安全性上讲，**不加同步的懒汉式是线程不安全的**，比如，有两个线程，一个是线程 A，一个是线程 B，它们同时调用 getInstance 方法，那就可能导致并发问题。如下示例：

```
public static  Singleton getInstance(){
    if(instance == null){
        instance = new Singleton();
    }
    return instance;
}
```

B 线程运行到这句话，正在进行判断（指向 `public static  Singleton getInstance(){`）

A 线程已经运行到这里了，还没有执行完下面一句话；而此时 B 线程运行到上面了，正在进行判断

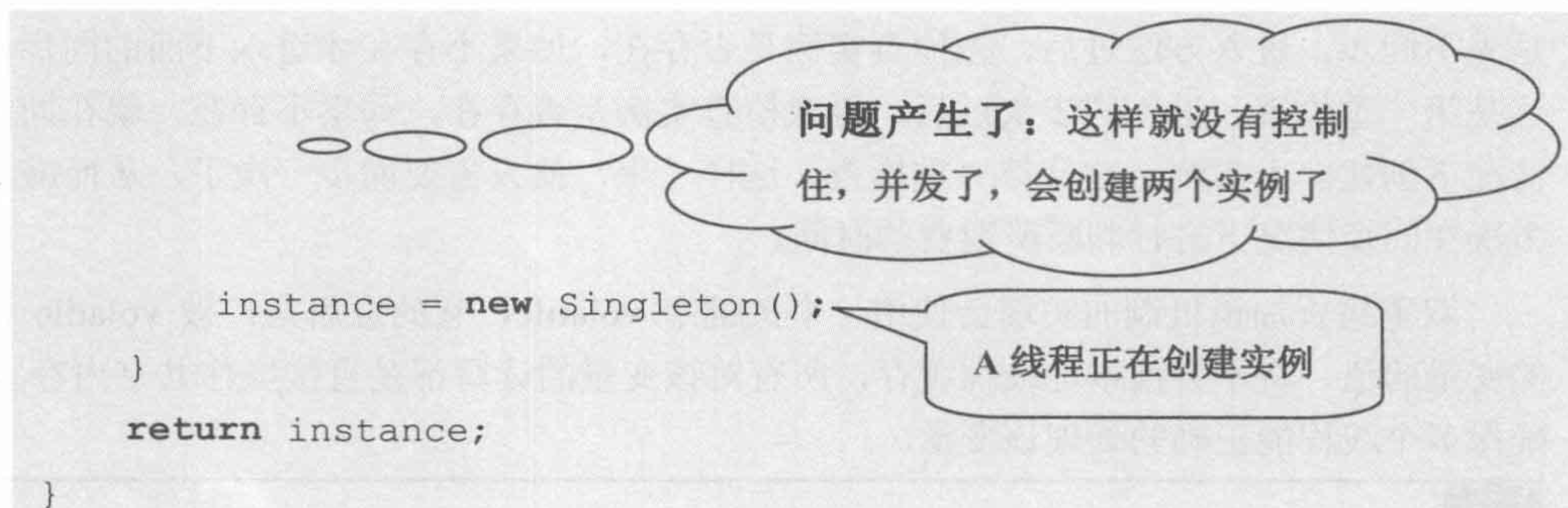
程序继续运行，两个线程都向前走了一步，如下：

```
public static  Singleton getInstance(){
    if(instance == null){
```

1：由于 B 线程运行较快，一下就判断出 instance==null，为 true

2：而此时 A 线程正在创建实例，也就是正运行 new Singleton()

3：但是 B 线程已经判断完了，也进入到这里了



可能有些朋友会觉得文字描述还是不够直观，再来画个图说明一下，如图 5.4 所示。

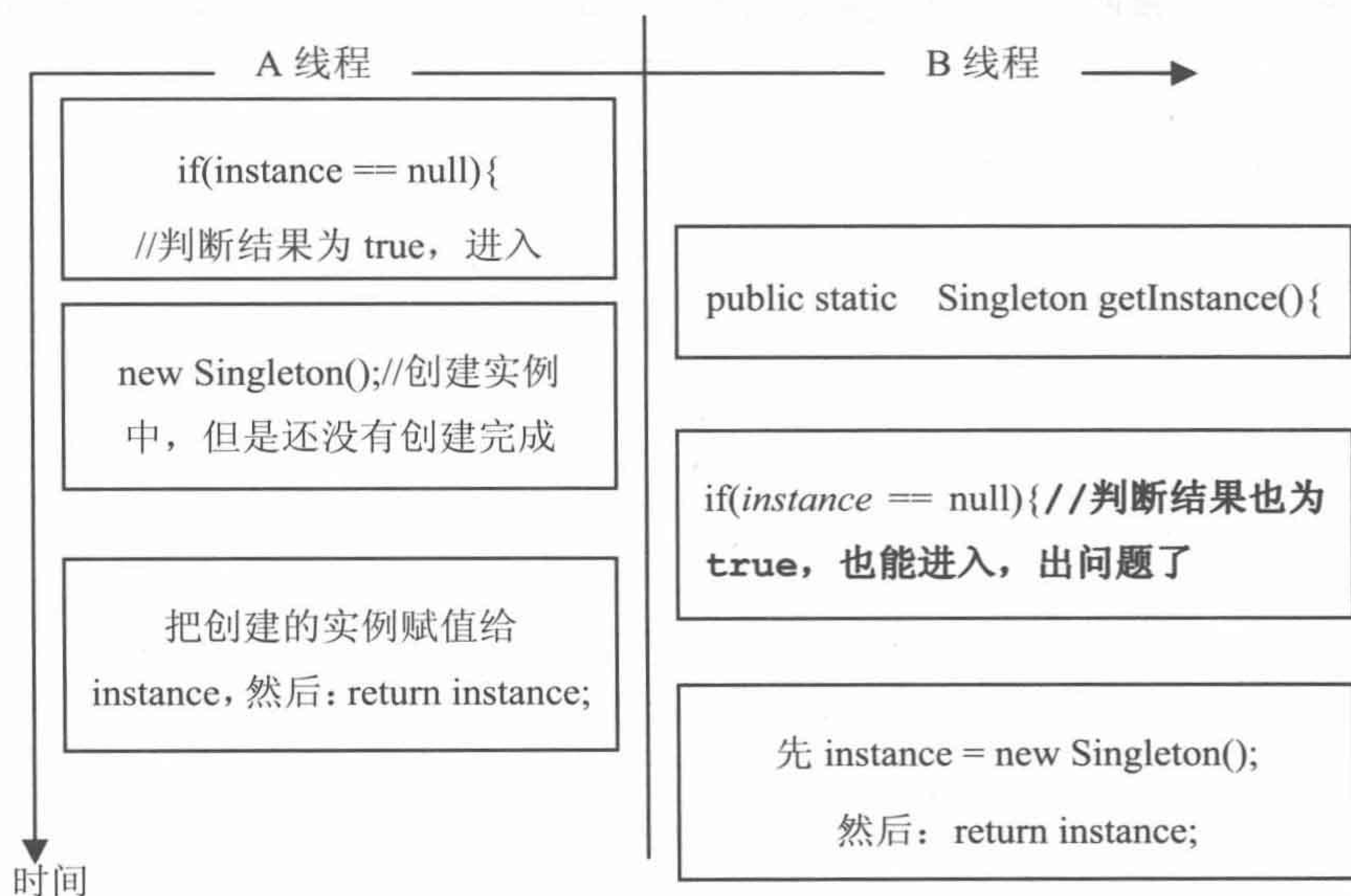


图 5.4　懒汉式单例的线程问题示意图

通过图 5.4 的分解描述，明显地看出，当 A、B 线程并发的情况下，会创建出两个实例来，也就是单例的控制在并发情况下失效了。

（2）**饿汉式是线程安全的**，因为虚拟机保证只会装载一次，在装载类的时候是不会发生并发的。

（3）如何实现懒汉式的线程安全呢？

当然懒汉式也是可以实现线程安全的，只要加上 synchronized 即可，如下：

```
public static synchronized Singleton getInstance(){}
```

但是这样一来，会降低整个访问的速度，而且每次都要判断。那么有没有更好的方式来实现呢？

（4）双重检查加锁

可以使用**“双重检查加锁”**的方式来实现，就可以既实现线程安全，又能够使性能不受到很大的影响。那么什么是“双重检查加锁”机制呢？

所谓双重检查加锁机制，指的是：并不是每次进入 getInstance 方法都需要同步，而

是先不同步，进入方法过后，先检查实例是否存在，如果不存在才进入下面的同步块，这是第一重检查。进入同步块过后，再次检查实例是否存在，如果不存在，就在同步的情况下创建一个实例，这是第二重检查。这样一来，就只需要同步一次了，从而减少了多次在同步情况下进行判断所浪费的时间。

双重检查加锁机制的实现会使用一个关键字 volatile，它的意思是：被 volatile 修饰的变量的值，将不会被本地线程缓存，所有对该变量的读写都是直接操作共享内存,从而确保多个线程能正确的处理该变量。

注意	**在 Java1.4 及以前版本中，很多 JVM 对于 volatile 关键字的实现有问题，会导致双重检查加锁的失败，因此双重检查加锁的机制只能用在 Java5 及以上的版本。**

看看代码可能会更加清楚些。示例代码如下：

```
public class Singleton {
    /**
     * 对保存实例的变量添加 volatile 的修饰
     */
    private volatile static Singleton instance = null;
    private Singleton(){
    }
    public static  Singleton getInstance(){
        //先检查实例是否存在，如果不存在才进入下面的同步块
        if(instance == null){
            //同步块，线程安全地创建实例
            synchronized(Singleton.class){
                //再次检查实例是否存在，如果不存在才真正地创建实例
                if(instance == null){
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

这种实现方式既可以实现线程安全地创建实例，而又不会对性能造成太大的影响。它只是在第一次创建实例的时候同步，以后就不需要同步了，从而加快了运行速度。

提示	**由于 volatile 关键字可能会屏蔽掉虚拟机中一些必要的代码优化，所以运行效率并不是很高。因此一般建议，没有特别的需要，不要使用。也就是说，虽然可以使用“双重检查加锁”机制来实现线程安全的单例，但并不建议大量采用，可以根据情况来选用。**

5.3.8　在 Java 中一种更好的单例实现方式

根据上面的分析，常见的两种单例实现方式都存在小小的缺陷，那么有没有一种方案，既能够实现延迟加载，又能够实现线程安全呢？

> **说明** **还真有高人想到这样的解决方案了，这个解决方案被称为 Lazy initialization holder class 模式，这个模式综合使用了 Java 的类级内部类和多线程缺省同步锁的知识，很巧妙地同时实现了延迟加载和线程安全。**

1. 相应的基础知识

先简单地看看类级内部类相关的知识。

- 什么是类级内部类？
  简单点说，类级内部类指的是，有 static 修饰的成员式内部类。如果没有 static 修饰的成员式内部类被称为对象级内部类。
- 类级内部类相当于其外部类的 static 成分，它的对象与外部类对象间不存在依赖关系，因此可直接创建。而对象级内部类的实例，是绑定在外部对象实例中的。
- 类级内部类中，可以定义静态的方法。在静态方法中只能够引用外部类中的静态成员方法或者成员变量。
- 类级内部类相当于其外部类的成员，只有在第一次被使用的时候才会被装载。

再来看看多线程缺省同步锁的知识。

大家都知道，在多线程开发中，为了解决并发问题，主要是通过使用 synchronized 来加互斥锁进行同步控制。但是在某些情况中，JVM 已经隐含地为您执行了同步，这些情况下就不用自己再来进行同步控制了。这些情况包括：

- 由静态初始化器（在静态字段上或 static{} 块中的初始化器）初始化数据时
- 访问 final 字段时
- 在创建线程之前创建对象时
- 线程可以看见它将要处理的对象时

2.　解决方案的思路

要想很简单地实现线程安全，可以采用静态初始化器的方式，它可以由 JVM 来保证线程的安全性。比如前面的饿汉式实现方式。但是这样一来，不是会浪费一定的空间吗？因为这种实现方式，会在类装载的时候就初始化对象，不管你需不需要。

如果现在有一种方法能够让类装载的时候不去初始化对象，那不就解决问题了？一种可行的方式就是采用类级内部类，在这个类级内部类里面去创建对象实例。这样一来，只要不使用到这个类级内部类，那就不会创建对象实例，从而同时实现延迟加载和线程安全。

看看代码示例可能会更清晰一些，示例代码如下：

```
public class Singleton {
    /**
     * 类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例
```

```
     * 没有绑定关系，而且只有被调用到时才会装载，从而实现了延迟加载
     */
    private static class SingletonHolder{
        /**
         * 静态初始化器，由 JVM 来保证线程安全
         */
        private static Singleton instance = new Singleton();
    }
    /**
     * 私有化构造方法
     */
    private Singleton(){
    }
    public static  Singleton getInstance(){
        return SingletonHolder.instance;
    }
}
```

仔细想想，是不是很巧妙呢！

当 getInstance 方法第一次被调用的时候，它第一次读取 SingletonHolder.instance，导致 SingletonHolder 类得到初始化；而这个类在装载并被初始化的时候，会初始化它的静态域，从而创建 Singleton 的实例，由于是静态的域，因此只会在虚拟机装载类的时候初始化一次，并由虚拟机来保证它的线程安全性。

这个模式的优势在于，getInstance 方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本。

5.3.9 单例和枚举

按照《高效 Java 第二版》中的说法：单元素的枚举类型已经成为实现 Singleton 的最佳方法。

为了理解这个观点，先来了解一点相关的枚举知识，这里只是强化和总结一下枚举的一些重要观点，更多基本的枚举的使用，请参看 Java 编程入门资料。

- Java 的枚举类型实质上是功能齐全的类，因此可以有自己的属性和方法。
- Java 枚举类型的基本思想是通过公有的静态 final 域为每个枚举常量导出实例的类。
- 从某个角度讲，枚举是单例的泛型化，本质上是单元素的枚举。

用枚举来实现单例非常简单，只需要编写一个包含单个元素的枚举类型即可。示例代码如下：

```
/**
 * 使用枚举来实现单例模式的示例
 */
```

```
public enum Singleton {
    /**
     * 定义一个枚举的元素，它就代表了 Singleton 的一个实例
     */
    uniqueInstance;

    /**
     * 示意方法，单例可以有自己的操作
     */
    public void singletonOperation(){
        //功能处理
    }
}
```

使用枚举来实现单实例控制会更加简洁，而且无偿地提供了序列化的机制，并由 JVM 从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。

5.3.10　思考单例模式

1. 单例模式的本质

> **单例模式的本质：控制实例数目。**

单例模式是为了控制在运行期间，某些类的实例数目只能有一个。可能有人就会思考，能不能控制实例数目为 2 个，3 个，或者是任意多个呢？目的都是一样的，节约资源啊，有些时候单个实例不能满足实际的需要，会忙不过来，根据测算，3 个实例刚刚好。也就是说，现在要控制实例数目为 3 个，怎么办呢？

其实思路很简单，就是利用上面通过 Map 来缓存实现单例的示例，进行变形，一个 Map 可以缓存任意多个实例。新的问题是，Map 中有多个实例，但是客户端调用的时候，到底返回哪一个实例呢，也就是实例的调度问题，我们只是想来展示设计模式，对于调度算法就不去深究了，做个最简单的循环返回就好可以了。示例代码如下：

```
/**
* 简单演示如何扩展单例模式，控制实例数目为 3 个
*/
public class OneExtend {
    /**
     * 定义一个缺省的 key 值的前缀
     */
    private final static String DEFAULT_PREKEY = "Cache";
```

```
    /**
     * 缓存实例的容器
     */
    private static Map<String,OneExtend> map =
                        new HashMap<String,OneExtend>();
    /**
     * 用来记录当前正在使用第几个实例，到了控制的最大数目，就返回从 1 开始
     */
    private static int num = 1;
    /**
     * 定义控制实例的最大数目
     */
    private final static int NUM_MAX = 3;
    private OneExtend(){}
    public static OneExtend getInstance(){
        String key = DEFAULT_PREKEY+num;
        OneExtend oneExtend = map.get(key);
        if(oneExtend==null){
            oneExtend = new OneExtend();
            map.put(key, oneExtend);
        }
        //把当前实例的序号加 1
        num++;
        if(num > NUM_MAX){
            //如果实例的序号已经达到最大数目了，那就重复从 1 开始获取
            num = 1;
        }
        return oneExtend;
    }

    public static void main(String[] args) {
        OneExtend t1 = getInstance ();
        OneExtend t2 = getInstance ();
        OneExtend t3 = getInstance ();
        OneExtend t4 = getInstance ();
        OneExtend t5 = getInstance ();
        OneExtend t6 = getInstance ();

        System.out.println("t1=="+t1);
        System.out.println("t2=="+t2);
```

缓存的体现，通过控制缓存的数据多少来控制实例数目

测试是否能满足功能要求

```
        System.out.println("t3=="+t3);
        System.out.println("t4=="+t4);
        System.out.println("t5=="+t5);
        System.out.println("t6=="+t6);
    }
}
```

测试一下，看看结果，如下：

```
t1==cn.javass.dp.singleton.example9.OneExtend@6b97fd
t2==cn.javass.dp.singleton.example9.OneExtend@1c78e57
t3==cn.javass.dp.singleton.example9.OneExtend@5224ee
t4==cn.javass.dp.singleton.example9.OneExtend@6b97fd
t5==cn.javass.dp.singleton.example9.OneExtend@1c78e57
t6==cn.javass.dp.singleton.example9.OneExtend@5224ee
```

第一个实例和第四个相同，第二个与第五个相同，第三个与第六个相同。也就是说一共只有三个实例，而且调度算法是从第一个依次取到第三个，然后回来继续从第一个开始取到第三个。

当然在这里我们不去考虑复杂的调度情况，也不去考虑何时应该创建新实例的问题。

注意　**这种实现方式同样是线程不安全的，需要处理，这里就不再展开去讲解了。**

2．何时选用单例模式

建议在如下情况时，选用单例模式。

当需要控制一个类的实例只能有一个，而且客户只能从一个全局访问点访问它时，可以选用单例模式，这些功能恰好是单例模式要解决的问题。

5.3.11　相关模式

很多模式都可以使用单例模式，只要这些模式中的某个类，需要控制实例为一个的时候，就可以很自然地使用上单例模式。比如抽象工厂方法中的具体工厂类就通常是一个单例。

读书笔记

第 6 章　工厂方法模式（Factory Method）

6.1 场景问题

6.1.1 导出数据的应用框架

考虑这样一个实际应用：实现一个导出数据的应用框架，来让客户选择数据的导出方式；并真正执行数据导出。

在一些实际的企业应用中，一个公司的系统往往分散在很多个不同的地方运行，比如各个分公司或者是门市点。公司既没有建立全公司专网的实力，但是又不愿意让业务数据实时地在广域网上传递，一个是考虑数据安全的问题，另一个是运行速度的问题。

这种系统通常会有一个折中的方案，那就是各个分公司内运行系统的时候是独立的，是在自己分公司的局域网内运行。每天业务结束的时候，各个分公司会导出自己的业务数据，然后把业务数据打包，通过网络传送给总公司，或是专人把数据送到总公司，然后由总公司进行数据导入和核算。

通常这种系统在导出数据上会有一些约定的方式，比如导出成文本格式、数据库备份形式、Excel 格式、Xml 格式等。

现在就来考虑实现这样一个应用框架。在继续之前，先来了解一些关于框架的知识。

6.1.2 框架的基础知识

1. 框架是什么

简单点说，**框架就是能完成一定功能的半成品软件。**

就其本质而言，框架是一个软件，而且是一个半成品的软件。所谓半成品，就是还不能完全实现用户需要的功能。框架只是实现用户需要的功能的一部分，还需要进一步加工，才能成为一个满足用户需要的、完整的软件。因此框架级的软件，它的主要客户是开发人员，而不是最终用户。

> 延伸
> **有些朋友会想，既然框架只是个半成品，那何必要去学习和使用框架呢，学习成本也不算小？那就是因为框架能完成一定的功能，也就是“框架已经完成的一定的功能”在吸引着开发人员，让大家去学习和使用框架。**

2. 框架能干什么

- **能完成一定功能，加快应用开发进度。**

  由于框架完成了一定的功能，而且通常是一些基础的、有难度的、通用的功能，这就避免我们在应用开发的时候完全从头开始，而是在框架已有的功能之上继续开发，也就是说会复用框架的功能，从而加快应用的开发进度。

- **给我们一个精良的程序架构。**

  框架定义了应用的整体结构，包括类和对象的分割、各部分的主要责任、类和对象怎么协作，以及控制流程等。

Java 界流行的框架，大多出自大师手笔，设计都很精良。基于这样的框架来开发，一般会遵循框架已经规划好的结构来进行开发，从而使开发应用程序的结构也相对变得精良了。

3. 对框架的理解

- **基于框架来开发，事情还是那些事情，只是看谁做的问题。**

对于应用程序和框架的关系，可以用一个图来简单描述一下，如图 6.1 所示。

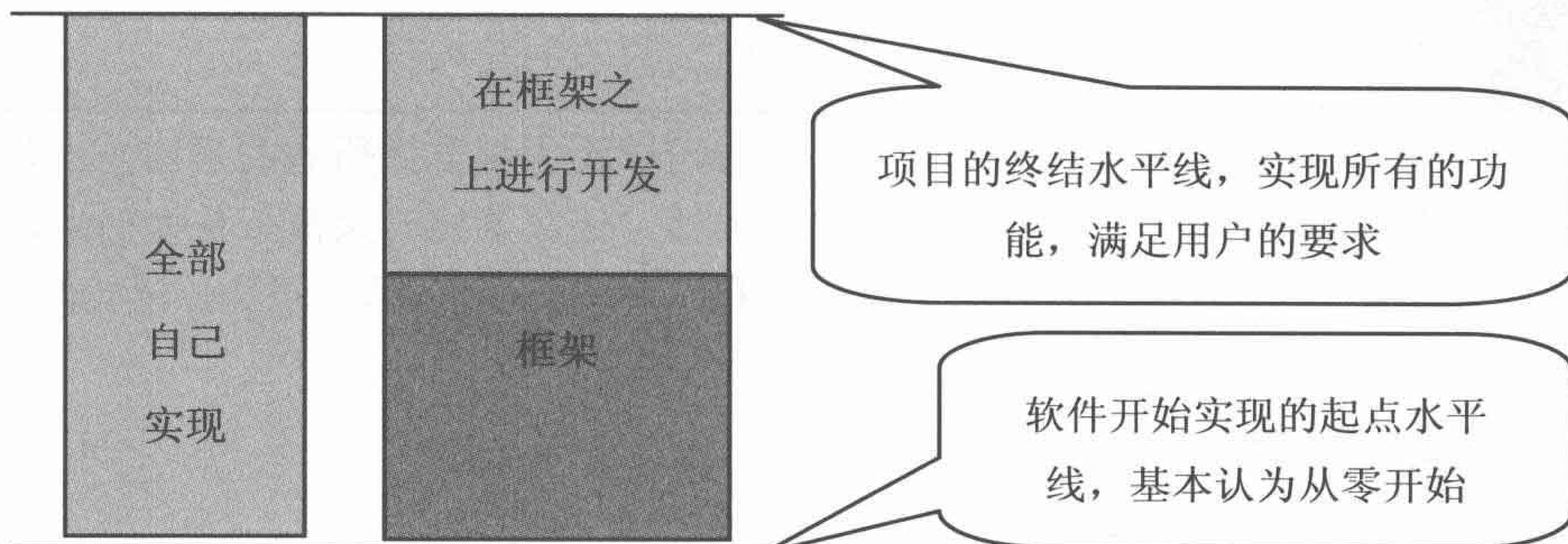


图 6.1　应用程序和框架的简单关系示意图

如果没有框架，那么客户要求的所有功能都由开发人员自己来开发，没问题，同样可以实现用户要求的功能，只是开发人员的工作多点。

如果有了框架，框架本身完成了一定的功能，那么框架已有的功能开发人员就可以不做了，开发人员只需要完成框架没有的功能，最后同样是完成客户要求的所有功能，但是开发人员的工作就减少了。

也就是说，基于框架来开发，软件要完成的功能并没有变化，还是客户要求的所有功能，也就是“事情还是那些事情”的意思。但是有了框架后，框架完成了一部分功能，然后开发人员再完成一部分功能，最后由框架和开发人员合起来完成了整个软件的功能，也就是看这些功能“由谁做”的问题。

- **基于框架开发，可以不去做框架所做的事情，但是应该明白框架在干什么，以及框架是如何实现相应功能的。**

事实上，在实际开发中，应用程序和框架的关系通常都不会像上面讲述的那样，分得那么清楚，更为普遍的是相互交互的。也就是应用程序做一部分工作，框架做另一部分工作，然后应用程序再做一部分工作，框架再做另一部分工作。如此交错，最后由应用程序和框架组合起来完成用户的功能要求。

也用个示意图来说明，如图 6.2 所示。

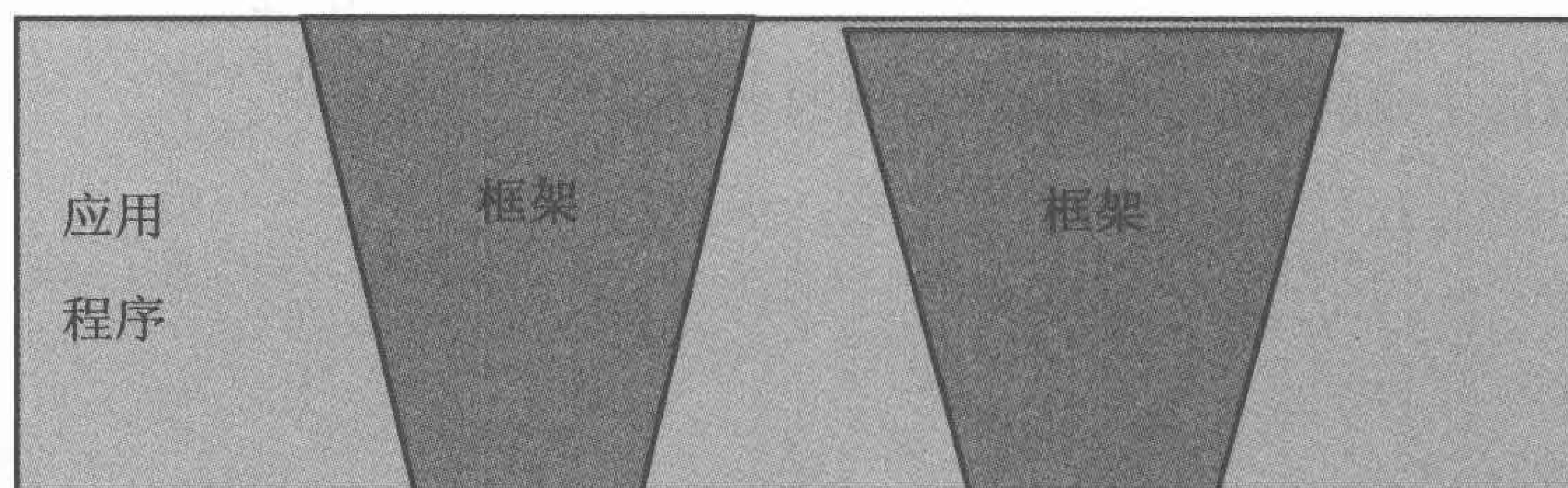


图 6.2　应用程序和框架的关系示意图

如果把这个由应用程序和框架组合在一起构成的矩形，当作最后完成的软件。试想一下，如果你不懂框架在干什么，相当于框架对你来讲是个黑盒，也就是相当于在图 6.2 中去掉框架的两块，会发现什么？没错，剩下的应用程序是支离破碎的，是相互分隔开来的。

延伸 **这会导致一个非常致命的问题，整个应用是如何运转起来的，你是不清楚的，也就是说对你而言，项目已经失控了，从项目管理的角度来讲，这是很危险的。**

因此，在基于框架开发的时候，虽然可以不去做框架所做的事情，但是应该搞明白框架在干什么，如果条件允许的话，还应该搞清楚框架是如何实现相应功能的，至少应该把大致的实现思路和实现步骤搞清楚，这样我们才能整体地掌控整个项目，才能尽量减少出现项目失控的情况。

4. 框架和设计模式的关系

1）设计模式比框架更抽象

框架已经是实现出来的软件了，虽然只是个半成品的软件，但毕竟是已经实现出来的了；而设计模式的重心还在于解决问题的方案上，也就是还停留在思想的层面上。因此设计模式比框架更为抽象。

2）设计模式是比框架更小的体系结构元素

如上所述，框架是已经实现出来的软件，并实现了一系列的功能，因此一个框架通常会包含多个设计模式的应用。

3）框架比设计模式更加特例化

框架是完成一定功能的半成品软件，也就是说，框架的目的很明确，就是要解决某一个领域的某些问题，那是很具体的功能。不同的领域实现出来的框架是不一样的。

而设计模式还停留在思想的层面，只要相应的问题适合用某个设计模式来解决，在不同的领域都可以应用。

因此框架总是针对特定领域的，而设计模式更加注重从思想上、方法上来解决问题，更加通用化。

6.1.3 有何问题

分析上面要实现的应用框架，不管用户选择什么样的导出格式，最后导出的都是一个文件，而且系统并不知道究竟要导出成为什么样的文件，因此应该有一个统一的接口来描述系统最后生成的对象，并操作输出的文件。

先把导出的文件对象的接口定义出来。示例代码如下：

```
/**
 * 导出的文件对象的接口
    */
public interface ExportFileApi {
```

```
    /**
     * 导出内容成为文件
     * @param data 示意：需要保存的数据
     * @return 是否导出成功
     */
    public boolean export(String data);
}
```

对于实现导出数据的业务功能对象，它应该根据需要来创建相应的 ExportFileApi 的实现对象，因为特定的 ExportFileApi 的实现是与具体的业务相关的。但是对于实现导出数据的业务功能对象而言，它并不知道应该创建哪一个 ExportFileApi 的实现对象，也不知道如何创建。

也就是说：对于实现导出数据的业务功能对象，它需要创建 ExportFileApi 的具体实例对象，但是它只知道 ExportFileApi 接口，而不知道其具体的实现，那该怎么办呢？

6.2 解决方案

6.2.1 使用工厂方法模式来解决问题

用来解决上述问题的一个合理的解决方案就是工厂方法模式（Factory Method）。那么什么是工厂方法模式呢？

1. 工厂方法模式的定义

> 定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method 使一个类的实例化延迟到其子类。

2. 应用工厂方法模式来解决问题的思路

仔细分析上面的问题，事实上在实现导出数据的业务功能对象里面，根本就不知道究竟要使用哪一种导出文件的格式，因此这个对象根本就不应该和具体的导出文件的对象耦合在一起，它只需要面向导出的文件对象的接口就可以了。

但是这样一来，又有新的问题产生了：接口是不能直接使用的，需要使用具体的接口实现对象的实例。

这不是自相矛盾吗？要求面向接口，不让和具体的实现耦合，但是又需要创建接口的具体实现对象的实例。怎么解决这个矛盾呢？

工厂方法模式的解决思路很有意思，那就是不解决，采取无为而治的方式：不是需要接口对象吗，那就定义一个方法来创建；可是事实上它自己是不知道如何创建这个接

口对象的，没有关系，定义成抽象方法就可以了，自己实现不了，那就让子类来实现，这样这个对象本身就可以只是面向接口编程，而无需关心到底如何创建接口对象了。

6.2.2 工厂方法模式的结构和说明

工厂方法模式的结构如图 6.3 所示。

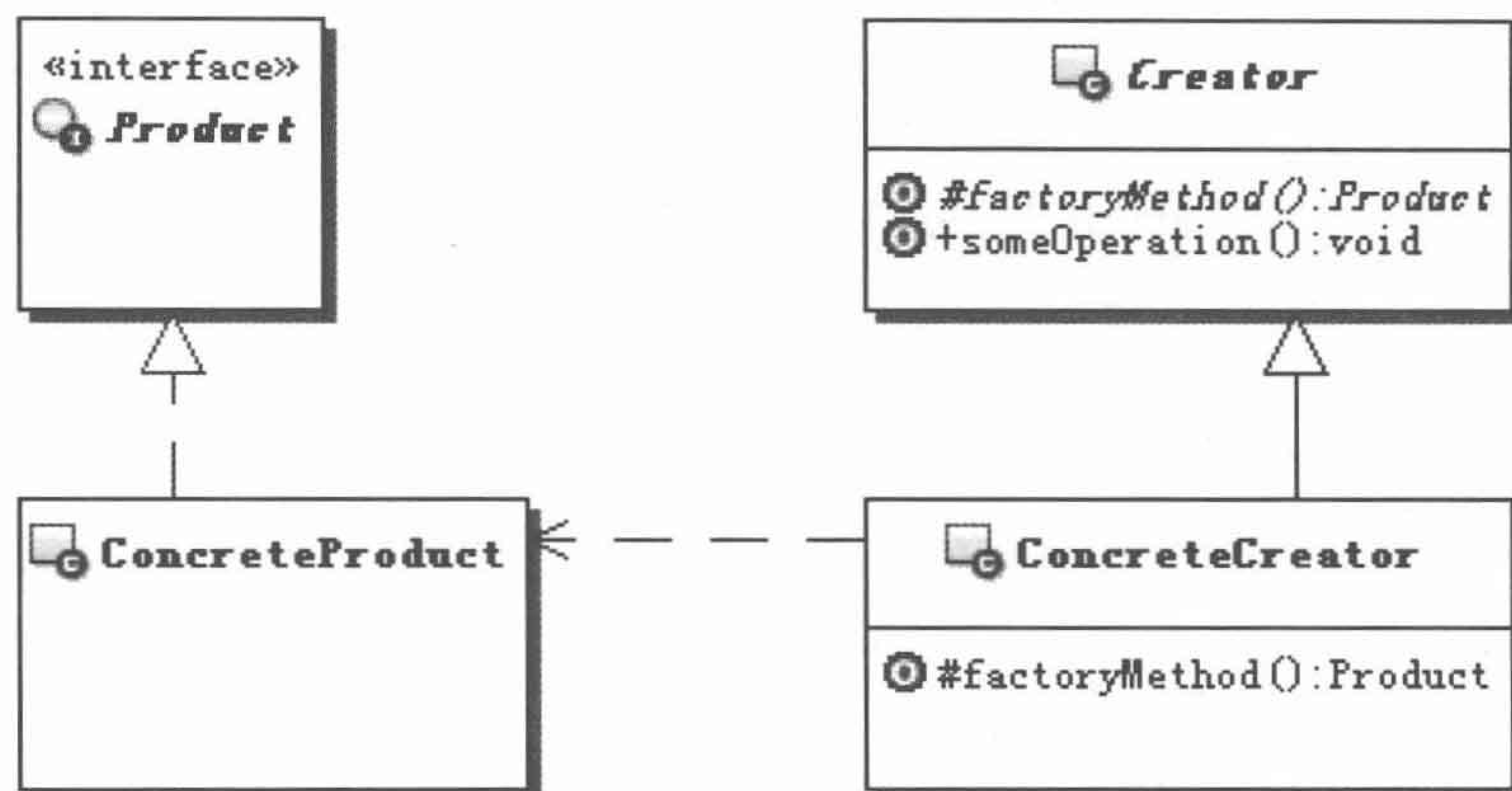


图 6.3 工厂方法模式结构示意图

- Product：定义工厂方法所创建的对象的接口，也就是实际需要使用的对象的接口。
- ConcreteProduct：具体的 Product 接口的实现对象。
- Creator：创建器，声明工厂方法，工厂方法通常会返回一个 Product 类型的实例对象，而且多是抽象方法。也可以在 Creator 里面提供工厂方法的默认实现，让工厂方法返回一个缺省的 Product 类型的实例对象。
- ConcreteCreator：具体的创建器对象，覆盖实现 Creator 定义的工厂方法，返回具体的 Product 实例。

6.2.3 工厂方法模式示例代码

（1）Product 定义的示例代码如下：

```
/**
 * 工厂方法所创建的对象的接口
 */
public interface Product {
    //可以定义 Product 的属性和方法
}
```

（2）Product 实现对象的示例代码如下：

```
/**
 * 具体的 Product 对象
 */
public class ConcreteProduct implements Product {
```

```
    //实现 Product 要求的方法
}
```

（3）创建器定义的示例代码如下：

```
/**
 * 创建器，声明工厂方法
 */
public abstract class Creator {
    /**
     * 创建 Product 的工厂方法
     * @return Product 对象
     */
    protected abstract Product factoryMethod();
    /**
     * 示意方法，实现某些功能的方法
     */
    public void someOperation() {
        //通常在这些方法实现中需要调用工厂方法来获取 Product 对象
        Product product = factoryMethod();
    }
}
```

（4）创建器实现对象的示例代码如下：

```
/**
 * 具体的创建器实现对象
 */
public class ConcreteCreator extends Creator {
    protected Product factoryMethod() {
        //重定义工厂方法，返回一个具体的 Product 对象
        return new ConcreteProduct();
    }
}
```

6.2.4　使用工厂方法模式来实现示例

要使用工厂方法模式来实现示例，先来按照工厂方法模式的结构，对应出哪些是被创建的 Product，哪些是 Creator。分析要求实现的功能，导出的文件对象接口 ExportFileApi 就相当于是 Product，而用来实现导出数据的业务功能对象就相当于 Creator。把 Product 和 Creator 分开后，就可以分别来实现它们了。

使用工厂模式来实现示例的程序结构如图 6.4 所示：

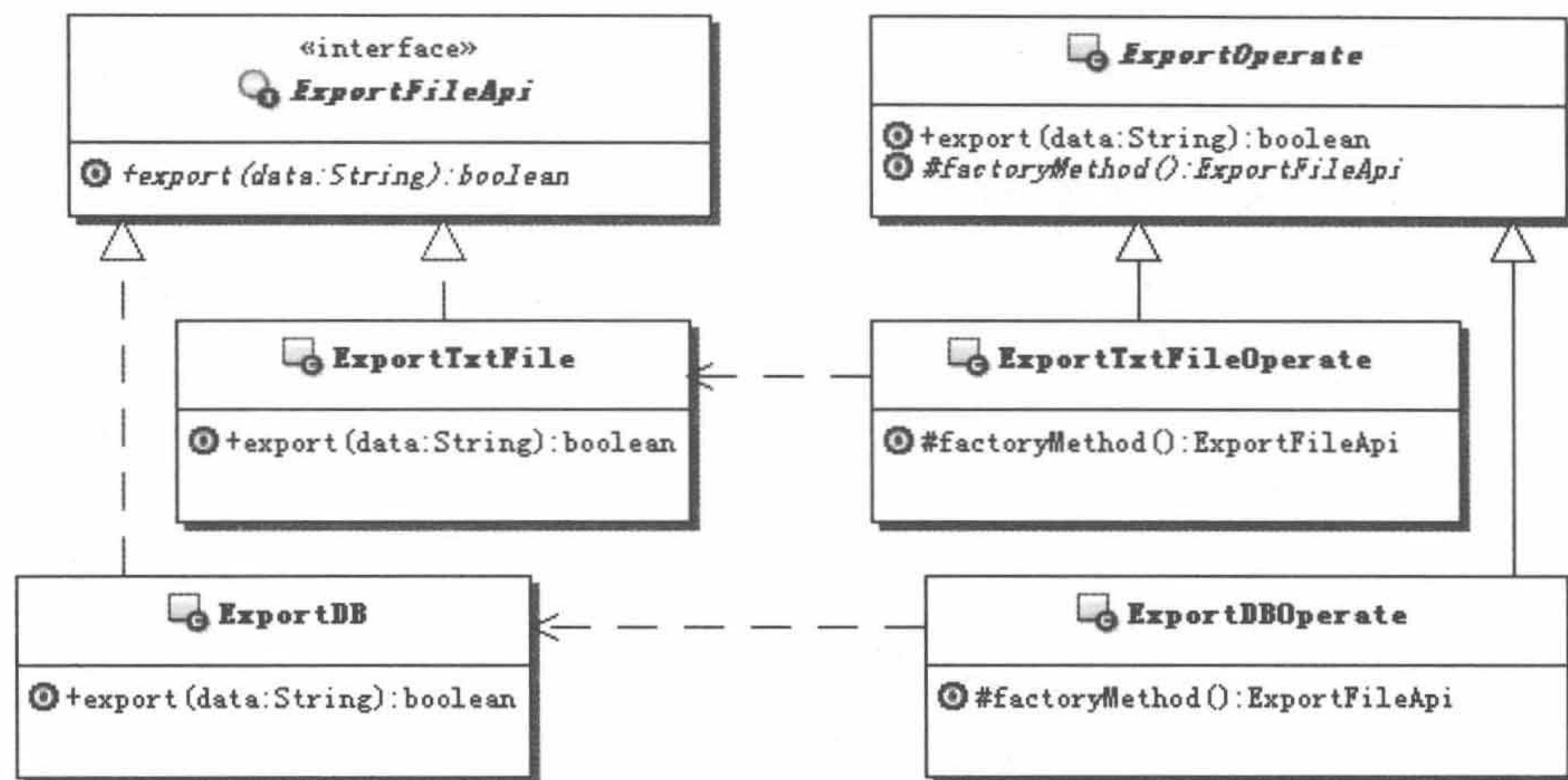


图 6.4　使用工厂方法模式来实现示例的程序结构示意图

下面一起来看看代码实现。

（1）导出的文件对象接口 ExportFileApi 的实现没有变化，这里就不再赘述了。

（2）接口 ExportFileApi 的实现。为了示例简单，只实现导出文本文件格式和数据库备份文件两种。

实现导出文本文件格式的。示例代码如下：

```
/**
 * 导出成文本文件格式的对象
 */
public class ExportTxtFile implements ExportFileApi{
    public boolean export(String data) {
        //简单示意一下，这里需要操作文件
        System.out.println("导出数据"+data+"到文本文件");
        return true;
    }
}
```

导出成数据库备份文件形式对象的示例代码如下：

```
/**
 * 导出成数据库备份文件形式的对象
 */
public class ExportDB implements ExportFileApi{
    public boolean export(String data) {
        //简单示意一下，这里需要操作数据库和文件
        System.out.println("导出数据"+data+"到数据库备份文件");
        return true;
    }
}
```

（3）实现 ExportOperate 的示例代码如下：

```
/**
 * 实现导出数据的业务功能对象
 */
public abstract class ExportOperate {
        /**
         * 导出文件
         * @param data 需要保存的数据
         * @return 是否成功导出文件
         */
        public boolean export(String data){
            //使用工厂方法
            ExportFileApi api = factoryMethod();
            return api.export(data);
        }
        /**
         * 工厂方法，创建导出的文件对象的接口对象
         * @return 导出的文件对象的接口对象
         */
        protected abstract ExportFileApi factoryMethod();
}
```

（4）加入了两个 Creator 实现。

创建导出成文本文件格式对象的示例代码如下：

```
/**
 * 具体的创建器实现对象，实现创建导出成文本文件格式的对象
 */
public class ExportTxtFileOperate extends ExportOperate{
        protected ExportFileApi factoryMethod() {
            //创建导出成文本文件格式的对象
            return new ExportTxtFile();
        }
}
```

创建导出成数据库备份文件形式对象的示例代码如下：

```
/**
   * 具体的创建器实现对象，实现创建导出成数据库备份文件形式的对象
   */
public class ExportDBOperate extends ExportOperate{
   protected ExportFileApi factoryMethod() {
        //创建导出成数据库备份文件形式的对象
   }
}
```

（5）客户端直接创建需要使用的 Creator 对象，然后调用相应的功能方法。示例代码如下：

```
public class Client {
        public static void main(String[] args) {
             //创建需要使用的Creator对象
             ExportOperate operate = new ExportDBOperate();
             //调用输出数据的功能方法
             operate.export("测试数据");
        }
}
```

运行结果如下：

```
导出数据测试数据到数据库备份文件
```

还可以修改客户端 new 的对象，切换成其他实现对象，试试看会发生什么。看来应用工厂方法模式是很简单的，对吧。

6.3 模 式 讲 解

6.3.1 认识工厂方法模式

1. 工厂方法模式的功能

工厂方法模式的主要功能是让父类在不知道具体实现的情况下，完成自身的功能调用；而具体的实现延迟到子类来实现。

这样在设计的时候，不用去考虑具体的实现，需要某个对象，把它通过工厂方法返回就好了，在使用这些对象实现功能的时候还是通过接口来操作，这类似于 IoC/DI 的思想，这个在后面将给大家稍详细点介绍。

2. 实现成抽象类

工厂方法的实现中，通常父类会是一个抽象类，里面包含创建所需对象的抽象方法，这些抽象方法就是工厂方法。

> 注意　**这里要注意一个问题，子类在实现这些抽象方法的时候，通常并不是真正地由子类来实现具体的功能，而是在子类的方法里面做选择，选择具体的产品实现对象。**

父类里面，通常会有使用这些产品对象来实现一定的功能的方法，而且这些方法所实现的功能通常都是公共的功能，不管子类选择了何种具体的产品实现，这些方法的功能总是能正确执行。

3. 实现成具体的类

也可以把父类实现成为一个具体的类。这种情况下，通常是在父类中提供获取所需对象的默认实现方法，这样即使没有具体的子类，也能够运行。

通常这种情况还是需要具体的子类来决定具体要如何创建父类所需要的对象。也把这种情况称为工厂方法为子类提供了挂钩。通过工厂方法，可以让子类对象来覆盖父类的实现，从而提供更好的灵活性。

4. 工厂方法的参数和返回

工厂方法的实现中，可能需要参数，以便决定到底选用哪一种具体的实现。也就是说通过在抽象方法里面传递参数，在子类实现的时候根据参数进行选择，看看究竟应该创建哪一个具体的实现对象。

一般工厂方法返回的是被创建对象的接口对象，当然也可以是抽象类或者一个具体的类的实例。

5. 谁来使用工厂方法创建的对象

这里首先要弄明白一件事情，就是谁在使用工厂方法创建的对象？

事实上，在工厂方法模式里面，应该是 Creator 中的其他方法在使用工厂方法创建的对象，虽然也可以把工厂方法创建的对象直接提供给 Creator 外部使用，但工厂方法模式的本意，是由 Creator 对象内部的方法来使用工厂方法创建的对象，也就是说，工厂方法一般不提供给 Creator 外部使用。

客户端应该使用 Creator 对象，或者是使用由 Creator 创建出来的对象。对于客户端使用 Creator 对象，这个时候工厂方法创建的对象，是 Creator 中的某些方法使用；对于使用那些由 Creator 创建出来的对象，这个时候工厂方法创建的对象，是构成客户端需要的对象的一部分。分别举例来说明。

1）客户端使用 Creator 对象的情况

比如前面的示例，对于“实现导出数据的业务功能对象”的类 ExportOperate，它有一个 export 的方法，在这个方法里面，需要使用具体的“导出的文件对象的接口对象” ExportFileApi，而 ExportOperate 是不知道具体的 ExportFileApi 实现的，那是怎么做的呢？就是定义了一个工厂方法，用来返回 ExportFileApi 的对象，然后 export 方法会使用这个

工厂方法来获取它所需要的对象，然后执行功能。

这个时候的客户端是怎么做的呢？这个时候客户端主要就是使用 ExportOperate 的实例来完成它想要完成的功能，也就是客户端使用 Creator 对象的情况。简单描述这种情况下的代码结构如下：

```
/**
 * 客户端使用 Creator 对象的情况下，Creator 的基本实现结构
 */
public abstract class Creator {
    /**
     * 工厂方法，一般不对外
     * @return 创建的产品对象
     */
    protected abstract Product factoryMethod();
    /**
     * 提供给外部使用的方法
     * 客户端一般使用 Creator 提供的这些方法来完成所需要的功能
     */
    public void someOperation(){
        //在这里使用工厂方法
        Product p = factoryMethod();
    }
}
```

2）客户端使用由 Creator 创建出来的对象

另外一种是由 Creator 向客户端返回由“工厂方法创建的对象”来构建的对象，这个时候工厂方法创建的对象，是构成客户端需要的对象的一部分。简单描述这种情况下的代码结构如下：

```
/**
 * 客户端使用 Creator 来创建客户端需要的对象的情况下，Creator 的基本实现结构
 */
public abstract class Creator {
    /**
     * 工厂方法，一般不对外，创建一个部件对象
     * @return 创建的产品对象，一般是另一个产品对象的部件
     */
    protected abstract Product1 factoryMethod1();
    /**
     * 工厂方法，一般不对外，创建一个部件对象
     * @return 创建的产品对象，一般是另一个产品对象的部件
     */
```

```
    protected abstract Product2 factoryMethod2();
    /**
     * 创建客户端需要的对象，客户端主要使用产品对象来完成所需要的功能
     * @return 客户端需要的对象
     */
    public Product createProduct(){
        //在这里使用工厂方法，得到客户端所需对象的部件对象
        Product1 p1 = factoryMethod1();
        Product2 p2 = factoryMethod2();

        //工厂方法创建的对象是创建客户端对象所需要的
        Product p = new ConcreteProduct();
        p.setProduct1(p1);
        p.setProduct2(p2);

        return p;
    }
}
```

> **提示**　**小结一下：在工厂方法模式里面，客户端要么使用 Creator 对象，要么使用 Creator 创建的对象，一般客户端不直接使用工厂方法。当然也可以直接把工厂方法暴露给客户端操作，但是一般不这么做。**

6．工厂方法模式的调用顺序示意图

由于客户端使用 Creator 对象有两种典型的情况，因此调用的顺序示意图也分为两种情况。

先看看客户端使用由 Creator 创建出来的对象情况的调用顺序示意图，如图 6.5 所示。

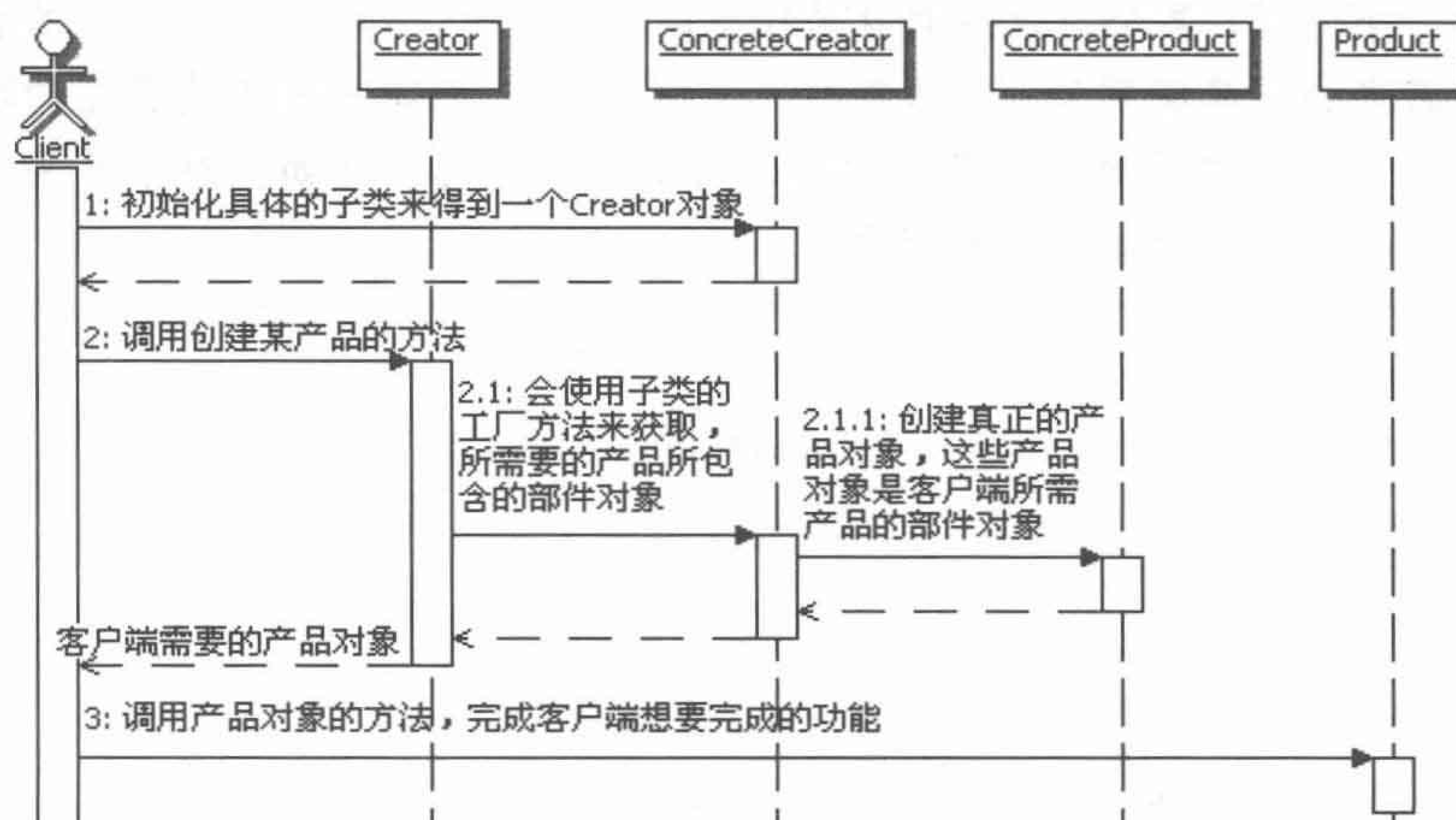


图 6.5　客户端使用由 Creator 创建出来的对象的调用顺序示意图

接下来看看客户端使用 Creator 对象时候的调用顺序示意图，如图 6.6 所示。

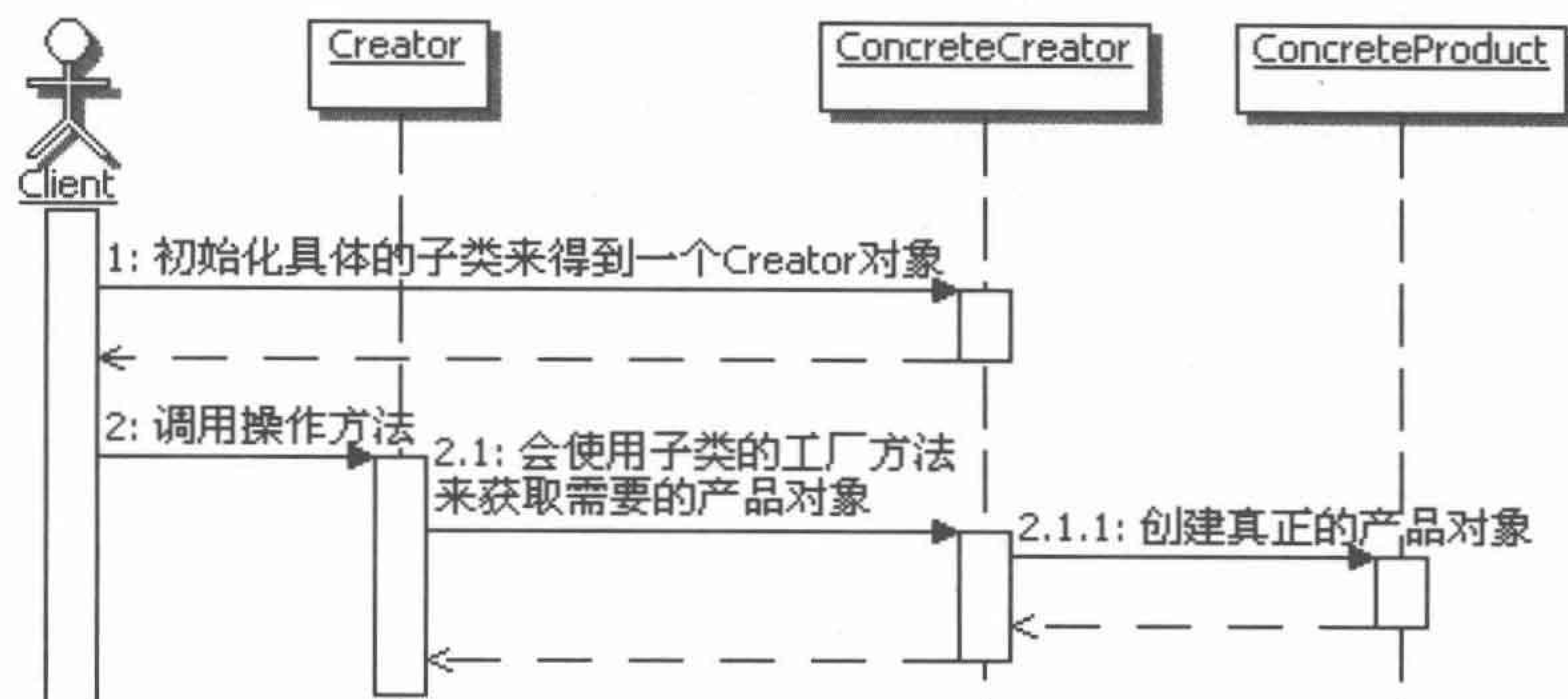


图 6.6　客户端使用 Creator 对象的调用顺序示意图

6.3.2　工厂方法模式与 IoC/DI

IoC——Inversion of Control，控制反转。

DI——Dependency Injection，依赖注入。

1. 如何理解 IoC/DI

要想理解上面两个概念，就必须搞清楚如下的问题：

- 参与者都有谁？
- 依赖：谁依赖于谁？为什么需要依赖？
- 注入：谁注入于谁？到底注入什么？
- 控制反转：谁控制谁？控制什么？为何叫反转（有反转就应该有正转了）？
- 依赖注入和控制反转是同一概念吗？

下面就来简要地回答一下上述问题，把这些问题搞明白了，也就明白 IoC/DI 了。

（1）参与者都有谁：一般有三方参与者，一个是某个对象；另一个是 IoC/DI 的容器；还有一个是某个对象的外部资源。

解释　**解释一下名词，某个对象指的就是任意的、普通的 Java 对象；IoC/DI 的容器简单点说就是指用来实现 IoC/DI 功能的一个框架程序；对象的外部资源指的就是对象需要的，但是是从对象外部获取的，都统称为资源，比如，对象需要的其他对象，或者是对象需要的文件资源等。**

（2）谁依赖于谁：当然是某个对象依赖于 IoC/DI 的容器。

（3）为什么需要依赖：对象需要 IoC/DI 的容器来提供对象需要的外部资源。

（4）谁注入于谁：很明显是 IoC/DI 的容器注入某个对象。

（5）到底注入什么：就是注入某个对象所需要的外部资源。

（6）谁控制谁：当然是 IoC/DI 的容器来控制对象了。

（7）控制什么：主要是控制对象实例的创建。

（8）为何叫反转：反转是相对于正向而言的，那么什么算是正向的呢？考虑一下常

规情况下的应用程序，如果要在 A 里面使用 C，你会怎么做呢？当然是直接去创建 C 的对象，也就是说，在 A 类中主动去获取所需要的外部资源 C，这种情况被称为正向的。那么什么是反向呢？就是 A 类不再主动去获取 C，而是被动等待，等待 IoC/DI 的容器获取一个 C 的实例，然后反向地注入到 A 类中。

用图例来说明一下。

先看没有 IoC/DI 的时候，常规的 A 类使用 C 的示意图，如图 6.7 所示。

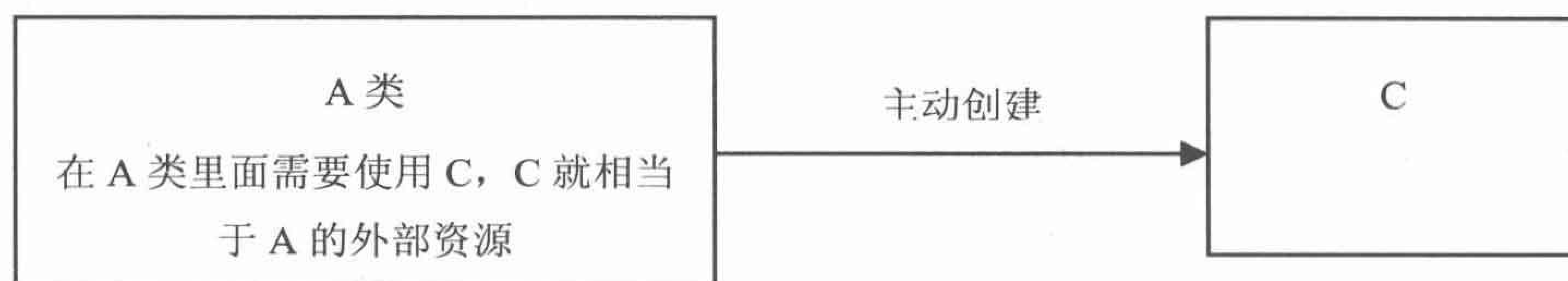


图 6.7　常规的 A 类使用 C 的示意图

当有了 IoC/DI 的容器后，A 类不再主动去创建 C 了，如图 6.8 所示。

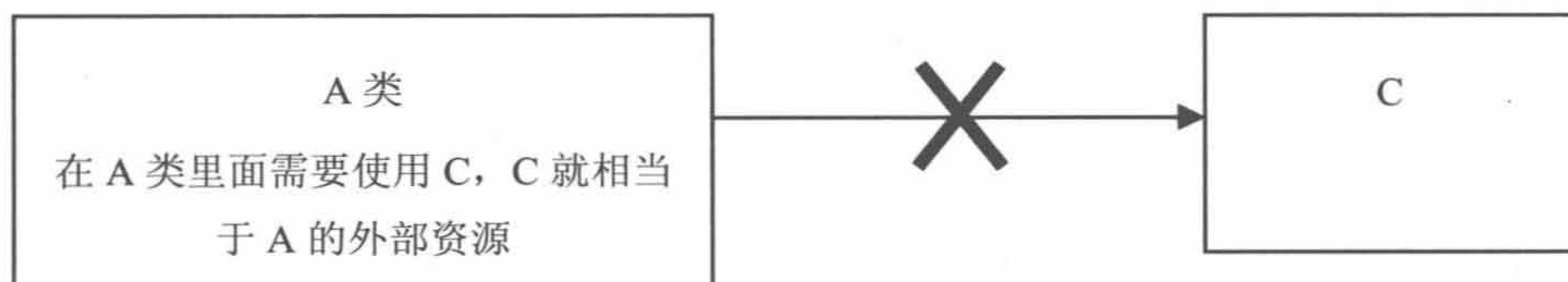


图 6.8　A 类不再主动创建 C

而是被动等待，等待 IoC/DI 的容器获取一个 C 的实例，然后反向地注入到 A 类中，如图 6.9 所示。

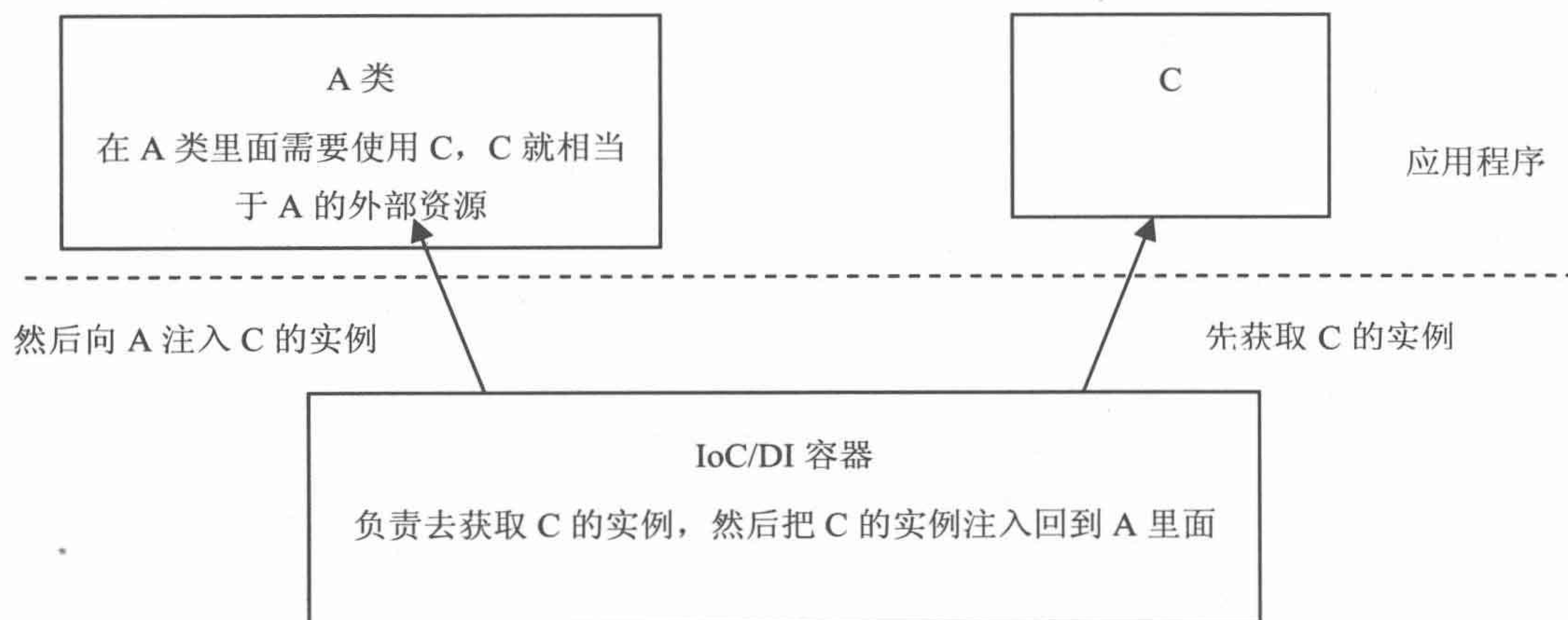


图 6.9　有了 IoC/DI 容器后的程序结构示意图

（9）依赖注入和控制反转是同一概念吗？

根据上面的讲述，应该能看出来，依赖注入和控制反转是对同一件事情的不同描述。从某个方面讲，就是它们描述的角度不同。依赖注入是从应用程序的角度去描述，可以把依赖注入描述得完整点：**应用程序依赖容器创建并注入它所需要的外部资源**；而控制反转是从容器的角度去描述，描述得完整点就是：**容器控制应用程序，由容器反向地向应用程序注入其所需要的外部资源**。

延伸

小结：其实 IoC/DI 对编程带来的最大改变不是在代码上，而是在思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在 IoC/DI 思想中，应用程序就变成被动的了，被动地等待 IoC/DI 容器来创建并注入它所需要的资源了。

这么小小的一个改变其实是编程思想的一个大进步，这样就有效地分离了对象和它所需要的外部资源，使得它们松散耦合，有利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

2. 工厂方法模式和 IoC/DI 的关系

从某个角度讲，工厂方法模式和 IoC/DI 的思想很类似。

上面讲了，有了 IoC/DI 后，应用程序就不再主动了，而是被动地等待由容器来注入资源。那么在编写代码的时候，一旦要用到外部资源，就会开一个窗口，让容器能注入进来，也就是提供给容器使用的注入的途径，当然这不是我们的重点，就不去细细讲解了，用 setter 注入来示例一下，使用 IoC/DI 的示例代码如下：

```
public class A {
    /**
     * 等待被注入进来
     */
    private C c = null;
    /**
     * 注入资源 C 的方法
     * @param c 被注入的资源
     */
    public void setC(C c){
        this.c = c;
    }
    public void t1(){
        //这里需要使用 C，可是又不让主动去创建 C 了，怎么办?
        //反正就要求从外部注入，这样更省心
        //自己不用管怎么获取 C，直接使用就好了
        c.tc();
    }
}
```

接口 C 的示例代码如下：

```
public interface C {
    public void tc();
}
```

从上面的示例代码可以看出，现在在 A 里面写代码的时候，凡是碰到了需要外部资

源，那么就提供注入的途径，要求从外部注入，自己只管使用这些对象。

再来看看工厂方法模式，如何实现上面同样的功能。为了区分，分别取名为 A1 和 C1。这个时候在 A1 里面要使用 C1 对象，也不是由 A1 主动去获取 C1 对象，而是创建一个工厂方法，类似于一个注入的途径；然后由子类，假设叫 A2 吧，由 A2 来获取 C1 对象，在调用的时候，替换掉 A1 的相应方法，相当于反向注入回到 A1 里面。示例代码如下：

```
public abstract class A1 {
    /**
     * 工厂方法，创建 C1，类似于从子类注入进来的途径
     * @return C1 的对象实例
     */
    protected abstract C1 createC1();
    public void t1(){
        //这里需要使用 C1 类，可是不知道究竟是用哪一个
        //也就不主动去创建 C1 了，怎么办？
        //反正会在子类里面实现，这里不用管怎么获取 C1，直接使用就好了
        createC1().tc();
    }
}
```

子类的示例代码如下：

```
public class A2 extends A1 {
    protected C1 createC1() {
        //真正的选择具体实现，并创建对象
        return new C2();
    }
}
```

C1 接口和前面的 C 接口是一样的，C2 这个实现类也是空的，只是演示一下，因此就不去展示它们的代码了。

提示	**仔细体会上面的示例，对比它们的实现，尤其是从思想层面上，会发现工厂方法模式和 IoC/DI 的思想是相似的，都是“主动变被动”，进行了“主从换位”，从而获得了更灵活的程序结构。**

6.3.3 平行的类层次结构

1. 平行的类层次结构的含义

简单点说，假如有两个类层次结构，其中一个类层次中的每个类在另一个类层次中都有一个对应的类的结构，就被称为平行的类层次结构。

举个例子来说，硬盘对象有很多种，如分成台式机硬盘和笔记本硬盘，在台式机硬盘的具体实现上面，又有希捷、西数等不同品牌的实现，同样在笔记本硬盘上，也有希捷、日立、IBM 等不同品牌的实现；硬盘对象具有自己的行为，如硬盘能存储数据，也能从硬盘上获取数据，不同的硬盘对象对应的行为对象是不一样的，因为不同的硬盘对象，它的行为的实现方式是不一样的。如果把硬盘对象和硬盘对象的行为分开描述，那么就构成了如图 6.10 所示的结构。

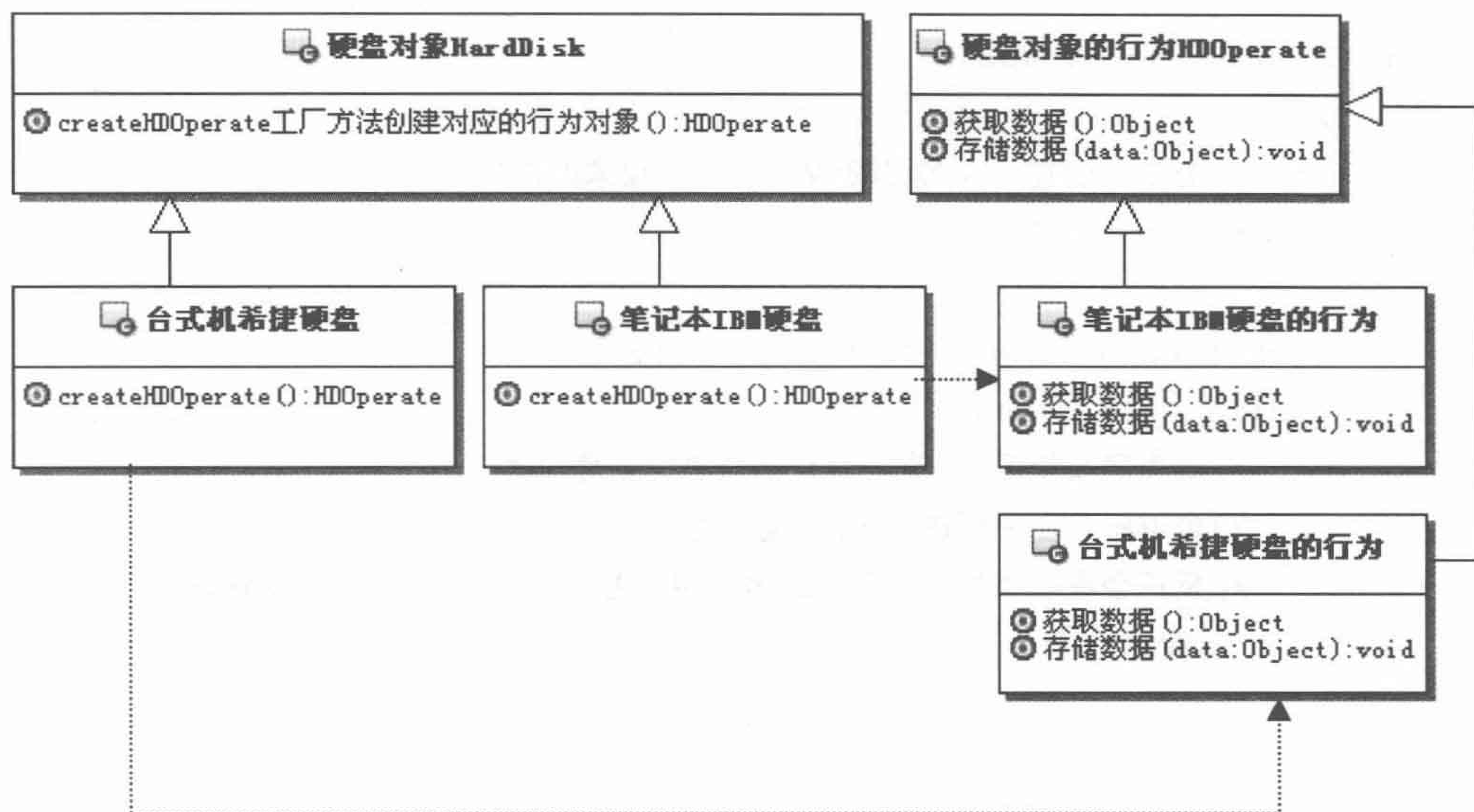


图 6.10　平行的类层次结构示意图

硬盘对象是一个类层次，硬盘的行为也是一个类层次，而且两个类层次中的类是对应的。台式机希捷硬盘对象就对应着硬盘行为里面的台式机希捷硬盘的行为；笔记本 IBM 硬盘就对应着笔记本 IBM 硬盘的行为，这就是一种典型的平行的类层次结构。

这种平行的类层次结构用来干什么呢？主要用来把一个类层次中的某些行为分离出来，让类层次中的类把原本属于自己的职责，委托给分离出来的类去实现，从而使得类层次本身变得更简单，更容易扩展和复用。

一般来讲，分离出去的这些类的行为，会对应着类层次结构来组织，从而形成一个新的类层次结构，相当于原来对象行为的类层次结构，而这个层次结构和原来的类层次结构是存在对应关系的，因此被称为平行的类层次结构。

2. 工厂方法模式和平行的类层次结构的关系

可以使用工厂方法模式来连接平行的类层次。

如图 6.10 所示，在每个硬盘对象里面，都有一个工厂方法 createHDOperate，通过这个工厂方法，客户端就可以获取一个和硬盘对象相对应的行为对象。在硬盘对象的子类里面，会覆盖父类的工厂方法 createHDOperate，以提供与自身相对应的行为对象，从而自然地把两个平行的类层次连接起来使用。

6.3.4 参数化工厂方法

所谓参数化工厂方法指的就是：**通过给工厂方法传递参数，让工厂方法根据参数的不同来创建不同的产品对象**，这种情况就被称为参数化工厂方法。当然工厂方法创建的不同的产品必须是同一个 Product 类型的。

来改造前面的示例，现在由一个工厂方法来创建 ExportFileApi 这个产品的对象，但是 ExportFileApi 接口的具体实现很多，为了方便创建的选择，直接从客户端传入一个参数，这样在需要创建 ExportFileApi 对象的时候，就把这个参数传递给工厂方法，让工厂方法来实例化具体的 ExportFileApi 实现对象。

还是看看代码示例会比较清楚。

（1）先来看 Product 的接口，就是 ExportFileApi 接口，和前面的示例相比没有任何变化，只是为了方便大家查看，这里重复一下。示例代码如下：

```
/**
 * 导出的文件对象的接口
 */
public interface ExportFileApi {
    /**
     * 导出内容成为文件
     * @param data 示意：需要保存的数据
     * @return 是否导出成功
     */
    public boolean export(String data);
}
```

（2）同样提供保存成文本文件和保存成数据库备份文件的实现，和前面的示例相比没有任何变化。示例代码如下：

```
public class ExportTxtFile implements ExportFileApi{
    public boolean export(String data) {
        //简单示意一下，这里需要操作文件
        System.out.println("导出数据"+data+"到文本文件");
        return true;
    }
}
public class ExportDB implements ExportFileApi{
    public boolean export(String data) {
        //简单示意一下，这里需要操作数据库和文件
        System.out.println("导出数据"+data+"到数据库备份文件");
        return true;
    }
}
```

（3）接下来该看看 ExportOperate 类了，这个类的变化大致如下。

- ExportOperate 类中的创建产品的工厂方法，通常需要提供默认的实现，不再抽象了，也就是变成了正常方法。
- ExportOperate 类也不再定义成抽象类了，因为有了默认的实现，客户端可能需要直接使用这个对象。
- 设置一个导出类型的参数，通过 export 方法从客户端传入。

看看代码吧，示例代码如下：

```
/**
 * 实现导出数据的业务功能对象
 */
public class ExportOperate {
    /**
     * 导出文件
     * @param type 用户选择的导出类型
     * @param data 需要保存的数据
     * @return 是否成功导出文件
     */
    public boolean export(int type,String data){
        //使用工厂方法
        ExportFileApi api = factoryMethod(type);
        return api.export(data);
    }
    /**
     * 工厂方法，创建导出的文件对象的接口对象
     * @param type 用户选择的导出类型
     * @return 导出的文件对象的接口对象
     */
    protected ExportFileApi factoryMethod(int type){
        ExportFileApi api = null;
        //根据类型来选择究竟要创建哪一种导出文件对象
        if(type==1){
            api = new ExportTxtFile();
        }else if(type==2){
            api = new ExportDB();
        }
        return api;
    }
}
```


（4）此时的客户端非常简单，直接使用 ExportOperate 类。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建需要使用的Creator对象
        ExportOperate operate = new ExportOperate();
        //调用输出数据的功能方法，传入选择导出类型的参数
        operate.export(1,"测试数据");
    }
}
```

测试看看，然后修改一下客户端的参数，体会一下通过参数来选择具体的导出实现的过程。

> 提示 **这是一种很常见的参数化工厂方法的实现方式，但是也还是有把参数化工厂方法实现成为抽象的，这点要注意，并不是说参数化工厂方法就不能实现成为抽象类了。只是一般情况下，参数化工厂方法，在父类都会提供默认的实现。**

（5）扩展新的实现。

使用参数化工厂方法，扩展起来会非常容易，已有的代码都不会改变，只要新加入一个子类来提供新的工厂方法实现，然后在客户端使用这个新的子类即可。

这种实现方式还有一个有意思的功能，就是子类可以选择性覆盖，不想覆盖的功能还可以返回去让父类来实现，很有意思。

扩展一个导出成 xml 文件的示例代码如下：

```
/**
 * 导出成xml文件的对象
 */
public class ExportXml implements ExportFileApi{
    public boolean export(String data) {
        //简单示意一下
        System.out.println("导出数据"+data+"到XML文件");
        return true;
    }
}
```

然后扩展 ExportOperate 类，来加入新的实现。示例代码如下：

```
/**
 * 扩展ExportOperate对象，加入可以导出的XML文件
 */
public class ExportOperate2 extends ExportOperate{
    /**
     * 覆盖父类的工厂方法，创建导出的文件对象的接口对象
     * @param type 用户选择的导出类型
```

```
     * @return 导出的文件对象的接口对象
     */
    protected ExportFileApi factoryMethod(int type){
        ExportFileApi api = null;
        //可以全部覆盖，也可以选择自己感兴趣的覆盖
        //这里只想添加自己新的实现，其他的不管
        if(type==3){
            api = new ExportXml();
        }else{
            //其他的还是让父类来实现
            api = super.factoryMethod(type);
        }
        return api;
    }
}
```

看看此时的客户端，也非常简单，只是在变换传入的参数。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建需要使用的 Creator 对象
        ExportOperate operate = new ExportOperate2();
        //下面变换传入的参数来测试参数化工厂方法
        operate.export(1,"Test1");
        operate.export(2,"Test2");
        operate.export(3,"Test3");
    }
}
```

对应的测试结果如下：

```
导出数据 Test1 到文本文件
导出数据 Test2 到数据库备份文件
导出数据 Test3 到 XML 文件
```

通过上面的示例，好好体会一下参数化工厂方法的实现和带来的好处。

6.3.5 工厂方法模式的优缺点

工厂方法模式的优点

- 可以在不知具体实现的情况下编程

  工厂方法模式可以让你在实现功能的时候，如果需要某个产品对象，只需要使用产品的接口即可，而无需关心具体的实现。选择具体实现的任务延迟到子类去完成。

- 更容易扩展对象的新版本

  工厂方法给子类提供了一个挂钩，使得扩展新的对象版本变得非常容易。比如上面示例的参数化工厂方法实现中，扩展一个新的导出 xml 文件格式的实现，已有的代码都不会改变，只要新加入一个子类来提供新的工厂方法实现，然后在客户端使用这个新的子类即可。

> **提示**　**另外这里提到的挂钩，就是我们经常说的钩子方法（hook），这个会在后面讲模板方法模式的时候详细点说明。**

- 连接平行的类层次

  工厂方法除了创造产品对象外，在连接平行的类层次上也大显身手。这个在前面已经详细讲述了。

工厂方法模式的缺点

- 具体产品对象和工厂方法的耦合性。

  在工厂方法模式中，工厂方法是需要创建产品对象的，也就是需要选择具体的产品对象，并创建它们的实例，因此具体产品对象和工厂方法是耦合的。

6.3.6　思考工厂方法模式

1. 工厂方法模式的本质

> 工厂方法模式的本质：**延迟到子类来选择实现。**

仔细体会前面的示例，你会发现，工厂方法模式中的工厂方法，在真正实现的时候，一般是先选择具体使用哪一个具体的产品实现对象，然后创建这个具体产品对象的示例，最后就可以返回去了。也就是说，工厂方法本身并不会去实现产品接口，具体的产品实现是已经写好了的，工厂方法只要去选择实现就好了。

有些朋友可能会说，这不是跟简单工厂一样吗？

从本质上讲，它们确实是非常类似的，在具体实现上都是 “选择实现”。但是也存在不同点，简单工厂是直接在工厂类里面进行“选择实现”；而工厂方法会把这个工作延迟到子类来实现，工厂类里面使用工厂方法的地方是依赖于抽象而不是具体的实现，从而使得系统更加灵活，具有更好的可维护性和可扩展性。

其实如果把工厂模式中的 Creator 退化一下，只提供工厂方法，而且这些工厂方法还都提供默认的实现，那不就变成简单工厂了吗？比如把刚才示范参数化工厂方法的例子代码拿过来再简化一下，你就能看出来，写得跟简单工厂是差不多的。示例代码如下：

```
public class ExportOperate {
    /**
     * 导出文件
     * @param type 用户选择的导出类型
```

```
        * @param data 需要保存的数据
        * @return 是否成功导出文件
        */
    public boolean export(int type,String data){
        //使用工厂方法
        ExportFileApi api = factoryMethod(type);
        return api.export(data);
    }
    /**
     * 工厂方法，创建导出的文件对象的接口对象
     * @param type 用户选择的导出类型
     * @return 导出的文件对象的接口对象
     */
    protected ExportFileApi factoryMethod(int type){
        ExportFileApi api = null;
        //根据类型来选择究竟要创建哪一种导出文件对象
        if(type==1){
            api = new ExportTxtFile();
        }else if(type==2){
            api = new ExportDB();
        }
        return api;
    }
}
```

简化这个Creator，把这些都删除

留下的这个方法，如果把它修改成public static的，是不是就和简单工厂写得一样了

看完上述代码，会体会到简单工厂和工厂方法模式是有很大相似性的了吧，从某个角度来讲，可以认为简单工厂就是工厂方法模式的一种特例，因此它们的本质是类似的，也就不足为奇了。

2. 对设计原则的体现

工厂方法模式很好地体现了“依赖倒置原则”。

依赖倒置原则告诉我们“要依赖抽象，不要依赖于具体类”，简单点说就是：不能让高层组件依赖于低层组件，而且不管高层组件还是低层组件，都应该依赖于抽象。

比如前面的示例，实现客户端请求操作的ExportOperate就是高层组件；而具体实现数据导出的对象就是低层组件，比如ExportTxtFile、ExportDB；而ExportFileApi接口就相当于是那个抽象。

对于ExportOperate来说，它不关心具体的实现方式，它只是“面向接口编程”；对于具体的实现来说，它只关心自己“如何实现接口”所要求的功能。

那么倒置的是什么呢？倒置的是这个接口的“所有权”。事实上，ExportFileApi接口中定义的功能，都是由高层组件ExportOperate来提出的要求，也就是说接口中的功能，是高层组件需要的功能。但是高层组件只是提出要求，并不关心如何实现，而底层组件，就是来真正实现高层组件所要求的接口功能的。因此看起来，低层实现的接口的所有权并不在底层组件手中，而是倒置到高层组件去了。

3. 何时选用工厂方法模式

建议在以下情况中选用工厂方法模式。

- 如果一个类需要创建某个接口的对象，但是又不知道具体的实现，这种情况可以选用工厂方法模式，把创建对象的工作延迟到子类中去实现。
- 如果一个类本身就希望由它的子类来创建所需的对象的时候，应该使用工厂方法模式。

6.3.7　相关模式

- 工厂方法模式和抽象工厂模式

  这两个模式可以组合使用，具体的放到抽象工厂模式中去讲。

- 工厂方法模式和模板方法模式

  这两个模式外观类似，都有一个抽象类，然后由子类来提供一些实现，但是工厂方法模式的子类专注的是创建产品对象，而模板方法模式的子类专注的是为固定的算法骨架提供某些步骤的实现。

  这两个模式可以组合使用，通常在模板方法模式里面，使用工厂方法来创建模板方法需要的对象。

读书笔记

第 7 章　抽象工厂模式（Abstract Factory）

7.1 场景问题

7.1.1 选择组装电脑的配件

举个生活中常见的例子——组装电脑，我们在组装电脑的时候，通常需要选择一系列的配件，比如 CPU、硬盘、内存、主板、电源、机箱等。为了使讨论简单点，只考虑选择 CPU 和主板的问题。

事实上，在选择 CPU 的时候，面临一系列的问题，比如品牌、型号、针脚数目、主频等问题，只有把这些都确定下来，才能确定具体的 CPU。

同样，在选择主板的时候，也有一系列的问题，比如品牌、芯片组、集成芯片、总线频率等问题，也只有这些都确定了，才能确定具体的主板。

选择不同的 CPU 和主板，是每个客户在组装电脑的时候，向装机公司提出的要求，也就是我们每个人自己拟定的装机方案。

在最终确定这个装机方案之前，还需要整体考虑各个配件之间的兼容性，比如，CPU 和主板，如果 CPU 针脚数和主板提供的 CPU 插口不兼容，是无法组装的。也就是说，装机方案是有整体性的，里面选择的各个配件之间是有关联的。

对于装机工程师而言，他只知道组装一台电脑，需要相应的配件，但是具体使用什么样的配件，还得由客户说了算。也就是说装机工程师只是负责组装，而客户负责选择装配所需要的具体的配件。因此，当装机工程师为不同的客户组装电脑时，只需要按照客户的装机方案，去获取相应的配件，然后组装即可。

现在需要使用程序来把这个装机的过程，尤其是选择组装电脑配件的过程实现出来，该如何实现呢？

7.1.2 不用模式的解决方案

考虑客户的功能，需要选择自己需要的 CPU 和主板，然后告诉装机工程师自己的选择，接下来就等着装机工程师组装电脑了。

对装机工程师而言，只是知道 CPU 和主板的接口，而不知道具体实现，很明显可以用上简单工厂或工厂方法模式。为了简单，这里选用简单工厂。客户告诉装机工程师自己的选择，然后装机工程师会通过相应的工厂去获取相应的实例对象。

（1）下面来看看 CPU 和主板的接口。

CPU 接口定义的示例代码如下：

```
/**
 * CPU 的接口
 */
public interface CPUApi {
    /**
     * 示意方法，CPU 具有运算的功能
```

```
    */
    public void calculate();
}
```

再看看主板的接口定义。示例代码如下：

```
/**
 * 主板的接口
 */
public interface MainboardApi {
    /**
     * 示意方法，主板都具有安装 CPU 的功能
     */
    public void installCPU();
}
```

（2）下面来看看具体的 CPU 实现。

Intel CPU 实现的示例代码如下：

```
/**
 *Intel 的 CPU 实现
 */
public class IntelCPU implements CPUApi{
    /**
     * CPU 的针脚数目
     */
    private int pins = 0;
    /**
     * 构造方法，传入 CPU 的针脚数目
     * @param pins CPU 的针脚数目
     */
    public IntelCPU(int pins){
        this.pins = pins;
    }
    public void calculate() {
        System.out.println("now in Intel CPU,pins="+pins);
    }
}
```

再看看 AMD 的 CPU 实现。示例代码如下：

```
/**
 * AMD 的 CPU 实现
 */
```

```
public class AMDCPU implements CPUApi{
    /**
     * CPU 的针脚数目
     */
    private int pins = 0;
    /**
     * 构造方法，传入 CPU 的针脚数目
     * @param pins CPU 的针脚数目
     */
    public AMDCPU(int pins){
        this.pins = pins;
    }
    public void calculate() {
        System.out.println("now in AMD CPU,pins="+pins);
    }
}
```

（3）下面来看看具体的主板实现。

技嘉主板实现的示例代码如下：

```
/**
 * 技嘉的主板
 */
public class GAMainboard implements MainboardApi {
    /**
     * CPU 插槽的孔数
     */
    private int cpuHoles = 0;
    /**
     * 构造方法，传入 CPU 插槽的孔数
     * @param cpuHoles CPU 插槽的孔数
     */
    public GAMainboard(int cpuHoles){
        this.cpuHoles = cpuHoles;
    }
    public void installCPU() {
        System.out.println("now in GAMainboard,cpuHoles="
                                                   +cpuHoles);
    }
}
```

微星主板实现的示例代码如下：

```
/**
 * 微星的主板
 */
public class MSIMainboard implements MainboardApi{
    /**
     * CPU 插槽的孔数
     */
    private int cpuHoles = 0;
    /**
     * 构造方法，传入 CPU 插槽的孔数
     * @param cpuHoles CPU 插槽的孔数
     */
    public MSIMainboard(int cpuHoles){
        this.cpuHoles = cpuHoles;
    }
    public void installCPU() {
        System.out.println("now in MSIMainboard,cpuHoles="
                                                +cpuHoles);
    }
}
```

（4）下面来看看创建 CPU 和主板的工厂。

创建 CPU 工厂实现的示例代码如下：

```
/**
 * 创建 CPU 的简单工厂
 */
public class CPUFactory {
    /**
     * 创建 CPU 接口对象的方法
     * @param type 选择 CPU 类型的参数
     * @return CPU 接口对象的方法
     */
    public static CPUApi createCPUApi(int type){
        CPUApi cpu = null;
        //根据参数来选择并创建相应的 CPU 对象
        if(type==1){
            cpu = new IntelCPU(1156);
        }else if(type==2){
            cpu = new AMDCPU(939);
```

```
        }
        return cpu;
    }
}
```

创建主板工厂实现的示例代码如下：

```
/**
 * 创建主板的简单工厂
 */
public class MainboardFactory {
    /**
     * 创建主板接口对象的方法
     * @param type 选择主板类型的参数
     * @return 主板接口对象的方法
     */
    public static MainboardApi createMainboardApi(int type){
        MainboardApi mainboard = null;
        //根据参数来选择并创建相应的主板对象
        if(type==1){
            mainboard = new GAMainboard(1156);
        }else if(type==2){
            mainboard = new MSIMainboard(939);
        }
        return mainboard;
    }
}
```

（5）下面看看装机工程师实现的示例代码如下：

```
/**
 * 装机工程师的类
 */
public  class ComputerEngineer {
    /**
     * 定义组装机器需要的CPU
     */
    private CPUApi cpu= null;
    /**
     * 定义组装机器需要的主板
     */
    private MainboardApi mainboard = null;
    /**
```

```
     * 装机过程
     * @param cpuType 客户选择所需 CPU 的类型
     * @param mainboardType 客户选择所需主板的类型
     */
    public void makeComputer(int cpuType,int mainboardType){
        //1：首先准备好装机所需要的配件
        prepareHardwares(cpuType,mainboardType);
        //2：组装机器
        //3：测试机器
        //4：交付客户
    }
    /**
     * 准备装机所需要的配件
     * @param cpuType 客户选择所需 CPU 的类型
     * @param mainboardType 客户选择所需主板的类型
     */
    private void prepareHardwares(int cpuType,int mainboardType){
        //这里要去准备 CPU 和主板的具体实现，为了示例简单，这里只准备这两个
        //可是，装机工程师并不知道如何去创建，怎么办呢？

        //直接找相应的工厂获取
        this.cpu = CPUFactory.createCPUApi(cpuType);
        this.mainboard = MainboardFactory.createMainboardApi(
                                                    mainboardType);
        //测试一下配件是否好用
        this.cpu.calculate();
        this.mainboard.installCPU();
    }
}
```

组装机器的基本步骤

（6）看看此时的客户端，应该通过装机工程师来组装电脑，客户需要告诉装机工程师他选择的配件。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建装机工程师对象
        ComputerEngineer engineer = new ComputerEngineer();
        //告诉装机工程师自己选择的配件，让装机工程师组装电脑
        engineer.makeComputer(1,1);
    }
}
```

运行结果如下：

```
now in Intel CPU,pins=1156
now in GAMainboard,cpuHoles=1156
```

7.1.3 有何问题

看了上面的实现，会感觉到很简单，通过使用简单工厂来获取需要的CPU和主板对象，然后就可以组装电脑了。有何问题呢？

上面的实现，虽然通过简单工厂解决了：对于装机工程师，只知CPU和主板的接口，而不知道具体实现的问题。**但还有一个问题没有解决，什么问题呢？那就是这些CPU对象和主板对象其实是有关系的，是需要相互匹配的。**而在上面的实现中，并没有维护这种关联关系，CPU和主板是由客户随意选择的。这是有问题的。

比如在上面实现中的客户端，在调用makeComputer时，传入参数为(1,2)，试试看，运行结果就会如下：

```
now in Intel CPU,pins=1156
now in MSIMainboard,cpuHoles=939
```

观察上面的结果，就会看出问题。客户选择的CPU的针脚是1156针的，而选择的主板上的CPU插孔却只有939针，根本无法组装。这就是没有维护配件之间的关系造成的。

该怎么解决这个问题呢？

7.2 解决方案

7.2.1 使用抽象工厂模式来解决问题

用来解决上述问题的一个合理的解决方案就是抽象工厂模式。那么什么是抽象工厂模式呢？

1. 抽象工厂模式的定义

> 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

2. 应用抽象工厂模式来解决问题的思路

仔细分析上面的问题，其实有两个问题点，一个是只知道所需要的一系列对象的接口，而不知具体实现，或者是不知道具体使用哪一个实现；另外一个是这一系列对象是相关或者相互依赖的，也就是说既要创建接口的对象，还要约束它们之间的关系。

有朋友可能会想，工厂方法模式或者是简单工厂，不就可以解决只知接口而不知实现的问题吗？怎么这些问题又冒出来了呢？

注意	**请注意，这里要解决的问题和工厂方法模式或简单工厂解决的问题是有很大不同的，工厂方法模式或简单工厂关注的是单个产品对象的创建，比如创建 CPU 的工厂方法，它就只关心如何创建 CPU 的对象，而创建主板的工厂方法，就只关心如何创建主板对象。**

这里要解决的问题是，要创建一系列的产品对象，而且这一系列对象是构建新的对象所需要的组成部分，也就是这一系列被创建的对象相互之间是有约束的。

解决这个问题的一个解决方案就是抽象工厂模式。在这个模式里面，会定义一个抽象工厂，在里面虚拟地创建客户端需要的这一系列对象，所谓虚拟的就是定义创建这些对象的抽象方法，并不去真正地实现，然后由具体的抽象工厂的子类来提供这一系列对象的创建。这样一来可以为同一个抽象工厂提供很多不同的实现，那么创建的这一系列对象也就不一样了，也就是说，抽象工厂在这里起到一个约束的作用，并提供所有子类的一个统一外观，来让客户端使用。

7.2.2　抽象工厂模式的结构和说明

抽象工厂模式的结构如图 7.1 所示。

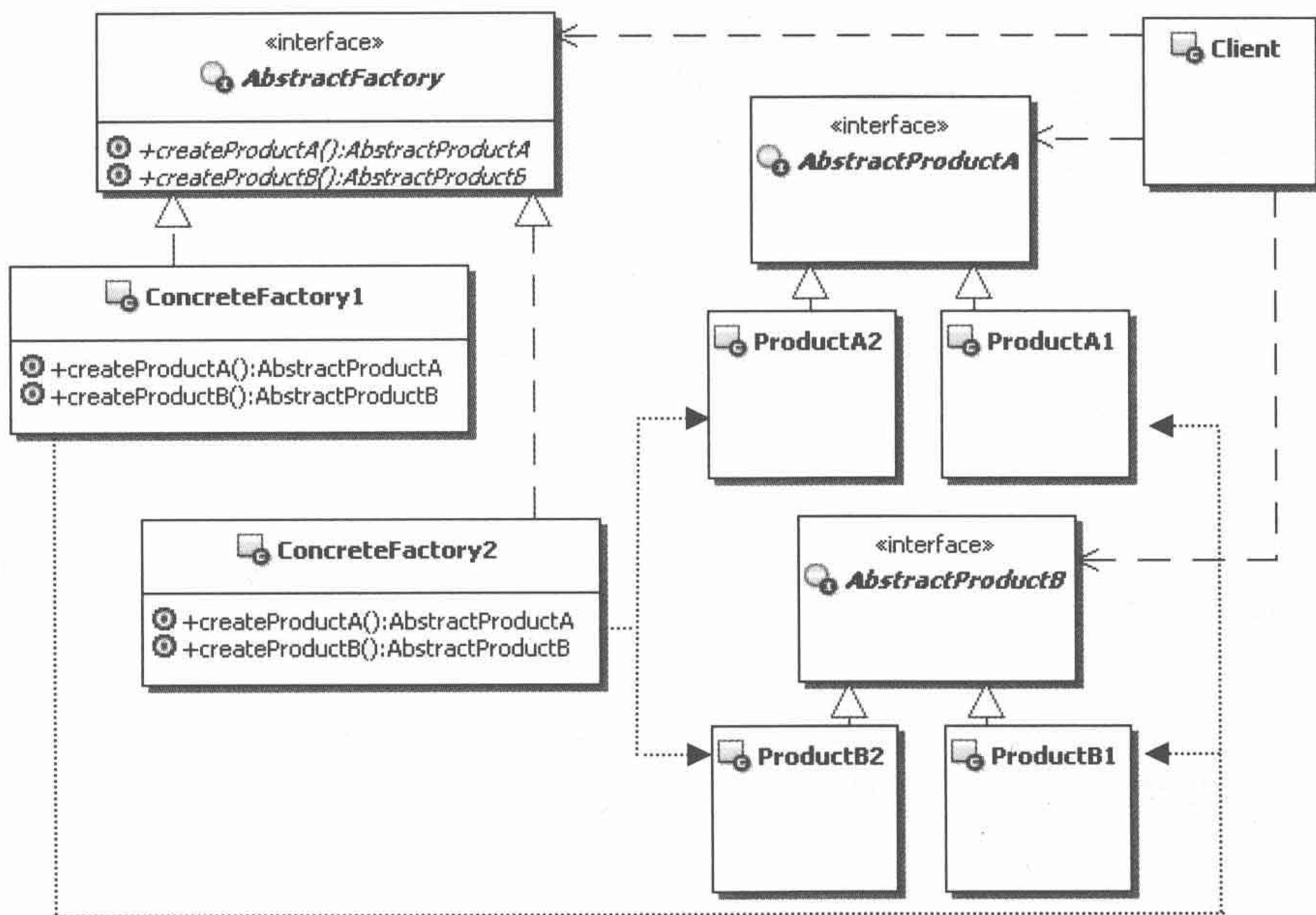


图 7.1　抽象工厂模式的结构示意图

- Abstract Factory：抽象工厂，定义创建一系列产品对象的操作接口。
- Concrete Factory：具体的工厂，实现抽象工厂定义的方法，具体实现一系列产品对象的创建。
- Abstract Product：定义一类产品对象的接口。
- Concrete Product：具体的产品实现对象，通常在具体工厂里面，会选择具体的产品实现对象，来创建符合抽象工厂定义的方法返回的产品类型的对象。
- Client：客户端，主要使用抽象工厂来获取一系列所需要的产品对象，然后面向这些产品对象的接口编程，以实现需要的功能。

7.2.3 抽象工厂模式示例代码

（1）先看看抽象工厂的定义。示例代码如下：

```
/**
 * 抽象工厂的接口，声明创建抽象产品对象的操作
 */
public interface AbstractFactory {
    /**
     * 示例方法，创建抽象产品 A 的对象
     * @return 抽象产品 A 的对象
     */
    public AbstractProductA createProductA();
    /**
     * 示例方法，创建抽象产品 B 的对象
     * @return 抽象产品 B 的对象
     */
    public AbstractProductB createProductB();
}
```

（2）接下来看看产品的定义，由于只是示意，并没有去定义具体的方法，示例代码如下：

```
/**
 * 抽象产品 A 的接口
 */
public interface AbstractProductA {
    //定义抽象产品 A 相关的操作
}
/**
 * 抽象产品 B 的接口
 */
public interface AbstractProductB {
```

```
    //定义抽象产品 B 相关的操作
}
```

（3）同样的，产品的各个实现对象也是空的。

实现产品 A 示例代码如下：

```
/**
 * 产品 A 的具体实现
 */
public class ProductA1 implements AbstractProductA {
    //实现产品 A 的接口中定义的操作
}
/**
 * 产品 A 的具体实现
 */
public class ProductA2 implements AbstractProductA {
    //实现产品 A 的接口中定义的操作
}
```

实现产品 B 的示例代码如下：

```
/**
 * 产品 B 的具体实现
 */
public class ProductB1 implements AbstractProductB {
    //实现产品 B 的接口中定义的操作
}
/**
 * 产品 B 的具体实现
 */
public class ProductB2 implements AbstractProductB {
    //实现产品 B 的接口中定义的操作
}
```

（4）再来看看具体的工厂的实现示意。示例代码如下：

```
/**
 * 具体的工厂实现对象，实现创建具体的产品对象的操作
 */
public class ConcreteFactory1 implements AbstractFactory {
    public AbstractProductA createProductA() {
        return new ProductA1();
    }
    public AbstractProductB createProductB() {
        return new ProductB1();
```

```
    }
}
/**
 * 具体的工厂实现对象，实现创建具体的产品对象的操作
 */
public class ConcreteFactory2 implements AbstractFactory {
    public AbstractProductA createProductA() {
        return new ProductA2();
    }
    public AbstractProductB createProductB() {
        return new ProductB2();
    }
}
```

（5）实现客户端的示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建抽象工厂对象
        AbstractFactory af = new ConcreteFactory1();
        //通过抽象工厂来获取一系列的对象，如产品 A 和产品 B
        af.createProductA();
        af.createProductB();
    }
}
```

7.2.4 使用抽象工厂模式重写示例

要使用抽象工厂模式来重写示例，先来看看如何使用抽象工厂模式来解决前面提出的问题。

装机工程师要组装电脑对象，需要一系列的产品对象，比如 CPU、主板等，于是创建一个抽象工厂给装机工程师使用，在这个抽象工厂里面定义抽象地创建 CPU 和主板的方法，这个抽象工厂就相当于一个抽象的装机方案，在这个装机方案里面，各个配件是能够相互匹配的。

每个装机的客户，会提出他们自己的具体装机方案，或者是选择已有的装机方案，相当于为抽象工厂提供了具体的子类，在这些具体的装机方案类里面，会创建具体的 CPU 和主板实现对象。

此时系统的结构如图 7.2 所示。

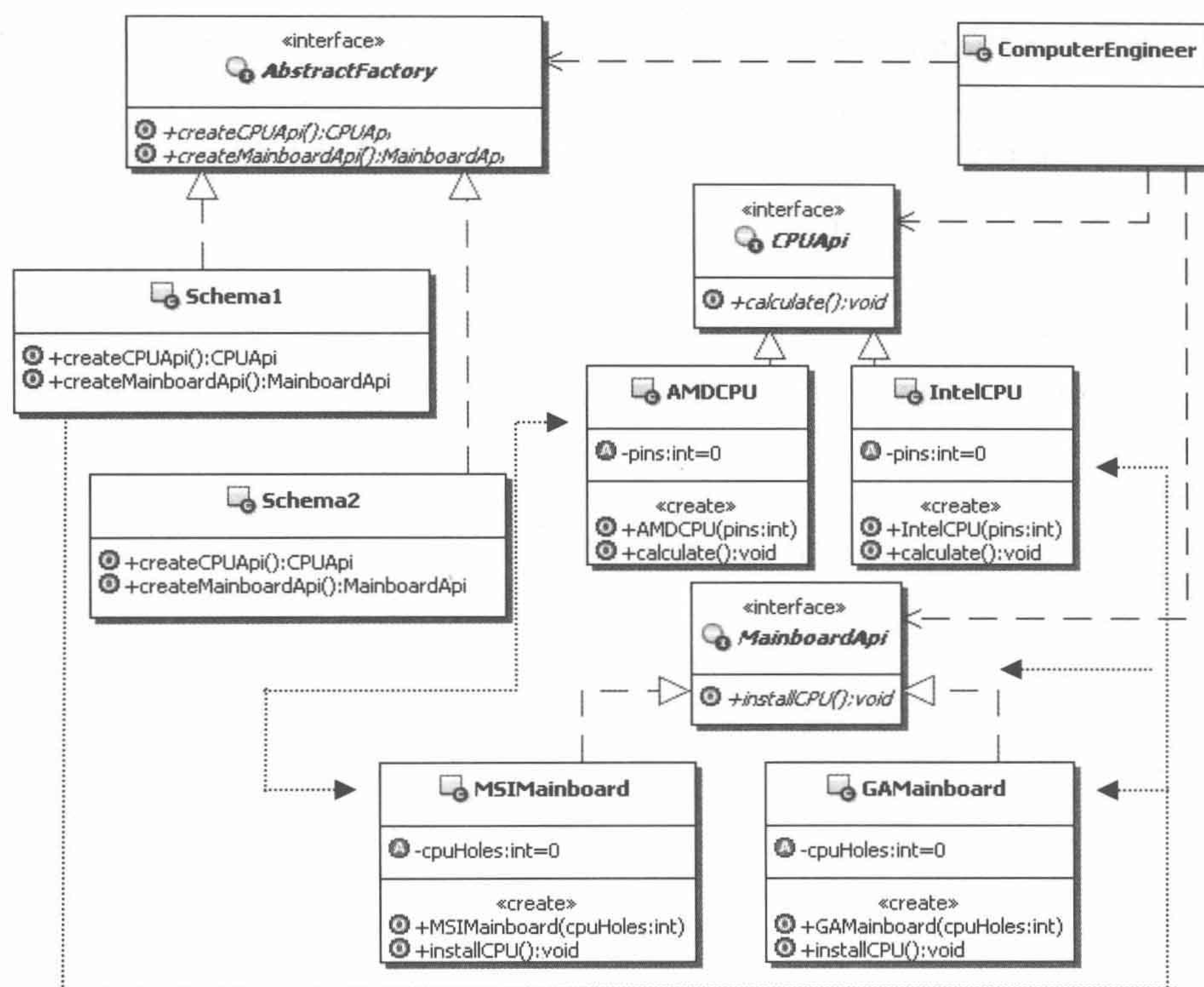


图 7.2　抽象工厂重写示例的结构示意图

虽然说是重写示例，但并不是前面写的都不要了，而是修改前面的示例，使它能更好地实现需要的功能。

（1）前面示例实现的 CPU 接口和 CPU 实现对象，还有主板的接口和实现对象，都不需要变化，这里就不再赘述了。

（2）前面示例中创建 CPU 的简单工厂和创建主板的简单工厂，都不再需要了，直接删除即可，这里也就不去管它了。

（3）看看新加入的抽象工厂的定义。示例代码如下：

```
/**
 * 抽象工厂的接口，声明创建抽象产品对象的操作
 */
public interface AbstractFactory {
    /**
     * 创建 CPU 的对象
     * @return CPU 的对象
     */
    public CPUApi createCPUApi();
    /**
     * 创建主板的对象
     * @return 主板的对象
     */
```

```
    public MainboardApi createMainboardApi();
}
```

（4）再看看抽象工厂的实现对象，也就是具体的装机方案对象。

先看看装机方案一的实现。示例代码如下：

```
/**
 * 装机方案一：Intel 的 CPU + 技嘉的主板
 * 这里创建 CPU 和主板对象的时候，是对应的，能匹配上的
 */
public class Schema1 implements AbstractFactory{
    public CPUApi createCPUApi() {
        return new IntelCPU(1156);
    }
    public MainboardApi createMainboardApi() {
        return new GAMainboard(1156);
    }
}
```

再看看装机方案二的实现。示例代码如下：

```
/**
 * 装机方案二：AMD 的 CPU + 微星的主板
 * 这里创建 CPU 和主板对象的时候，是对应的，能匹配上的
 */
public class Schema2 implements AbstractFactory{
    public CPUApi createCPUApi() {
        return new AMDCPU(939);
    }
    public MainboardApi createMainboardApi() {
        return new MSIMainboard(939);
    }
}
```

（5）下面来看看装机工程师类的实现。在现在的实现里面，装机工程师相当于使用抽象工厂的客户端，虽然是由真正的客户来选择和创建具体的工厂对象，但是使用抽象工厂的是装机工程师对象。

装机工程师类跟前面的实现相比，主要的变化是：从客户端不再传入选择 CPU 和主板的参数，而是直接传入客户选择并创建好的装机方案对象。这样就避免了单独去选择 CPU 和主板，客户要选就是一套，就是一个系列。示例代码如下：

```
/**
 * 装机工程师的类
 */
```

```
public  class ComputerEngineer {
    /**
     * 定义组装电脑需要的 CPU
     */
    private CPUApi cpu= null;
    /**
     * 定义组装电脑需要的主板
     */
    private MainboardApi mainboard = null;

    /**
     * 装机过程
     * @param schema 客户选择的装机方案
     */
    public void makeComputer(AbstractFactory schema){
        //1：首先准备好装机所需要的配件
        prepareHardwares(schema);
        //2：组装电脑
        //3：测试电脑
        //4：交付客户
    }
    /**
     * 准备装机所需要的配件
     * @param schema 客户选择的装机方案
     */
    private void prepareHardwares(AbstractFactory schema){
        //这里要去准备 CPU 和主板的具体实现，为了示例简单，这里只准备这两个
        //可是，装机工程师并不知道如何去创建，怎么办呢？

        //使用抽象工厂来获取相应的接口对象
        this.cpu = schema.createCPUApi();
        this.mainboard = schema.createMainboardApi();

        //测试一下配件是否好用
        this.cpu.calculate();
        this.mainboard.installCPU();
    }
}
```

（6）都定义好了，下面看看客户端如何使用抽象工厂。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建装机工程师对象
        ComputerEngineer engineer = new ComputerEngineer();
        //客户选择并创建需要使用的装机方案对象
        AbstractFactory schema = new Schema1();
        //告诉装机工程师自己选择的装机方案，让装机工程师组装电脑
        engineer.makeComputer(schema);
    }
}
```

运行一下，测试看看，是否能满足功能的要求。

如同前面的示例，定义了一个抽象工厂 AbstractFactory，在里面定义了创建 CPU 和主板对象的接口的方法，但是在抽象工厂里面，并没有指定具体的 CPU 和主板的实现，也就是无须指定它们具体的实现类。

CPU 和主板是相关的对象，是构建电脑的一系列相关配件，这个抽象工厂就相当于一个装机方案，客户选择装机方案的时候，一选就是一套，CPU 和主板是确定好的，不让客户分开选择，这就避免了出现不匹配的错误。

7.3 模式讲解

7.3.1 认识抽象工厂模式

1. 抽象工厂模式的功能

抽象工厂的功能是为一系列相关对象或相互依赖的对象创建一个接口。一定要注意，这个接口内的方法不是任意堆砌的，而是一系列相关或相互依赖的方法，比如上面例子中的 CPU 和主板，都是为了组装一台电脑的相关对象。

从某种意义上看，抽象工厂其实是一个产品系列，或者是产品簇。上面例子中的抽象工厂就可以看成是电脑簇，每个不同的装机方案，代表一种具体的电脑系列。

2. 实现成接口

AbstractFactory 在 Java 中通常实现成为接口，大家不要被名称误导了，以为是实现成为抽象类。当然，如果需要为这个产品簇提供公共的功能，也不是不可以把 AbstractFactory 实现成为抽象类，但一般不这么做。

3. 使用工厂方法

AbstractFactory 定义了创建产品所需要的接口，具体的实现是在实现类里面，通常在实现类里面就需要选择多种更具体的实现。所以 AbstractFactory 定义的创建产品的方法可以看成是工厂方法，而这些工厂方法的具体实现就延迟到了具体的工厂里面。也就是说使用工厂方法来实现抽象工厂。

4. 切换产品簇

由于抽象工厂定义的一系列对象通常是相关或者相互依赖的，这些产品对象就构成了一个产品簇，也就是抽象工厂定义了一个产品簇。

这就带来非常大的灵活性，切换一个产品簇的时候，只要提供不同的抽象工厂实现就可以了，也就是说现在是以产品簇作为一个整体被切换。

5. 抽象工厂模式的调用顺序示意图

抽象工厂模式的调用顺序如图 7.3 所示。

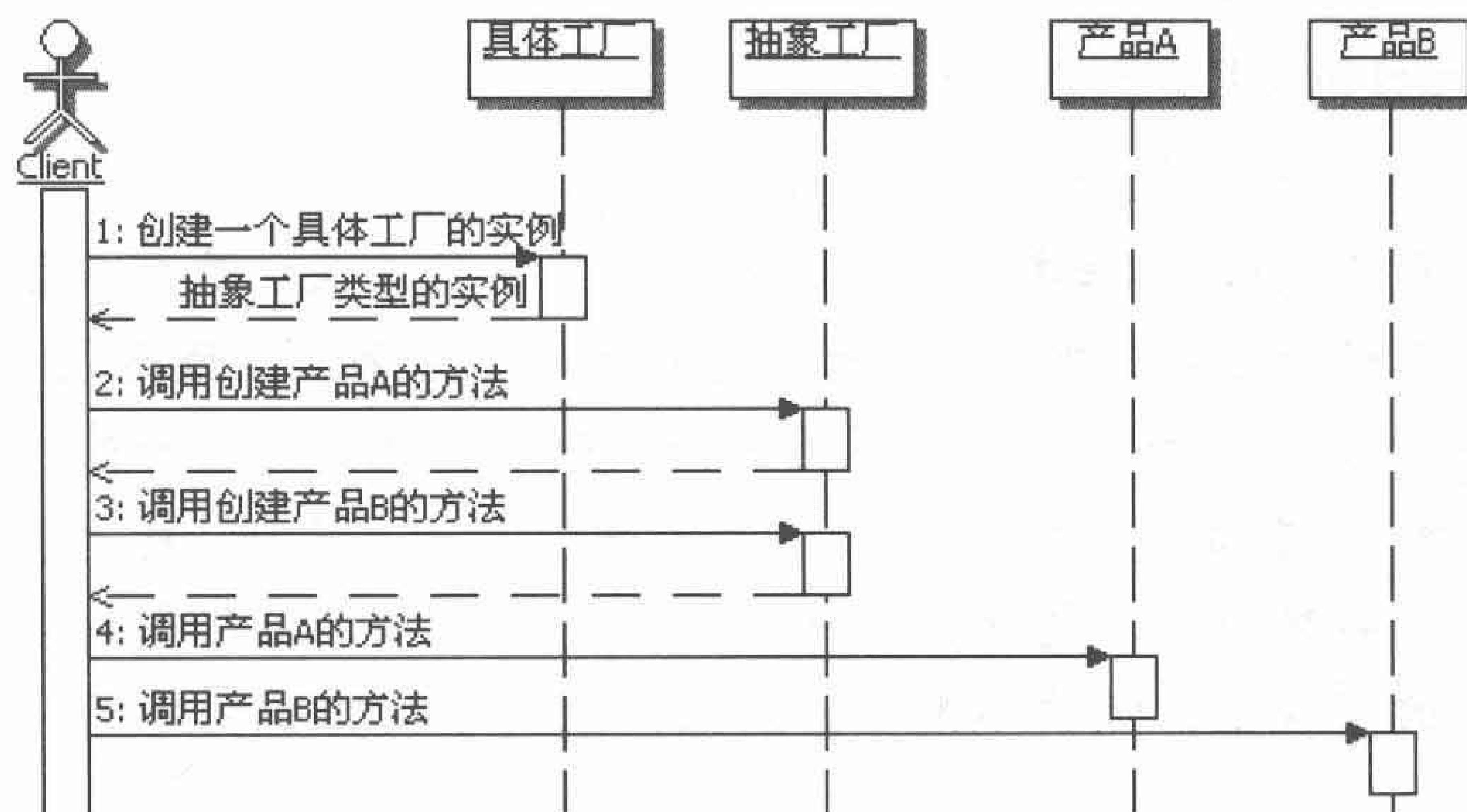


图 7.3　抽象工厂模式的调用顺序示意图

7.3.2　定义可扩展的工厂

在前面的示例中，抽象工厂为每一种它能创建的产品对象定义了相应的方法，比如创建 CPU 的方法和创建主板的方法等。

这种实现有一个麻烦，就是如果在产品簇中要新增加一种产品，比如现在要求抽象工厂除了能够创建 CPU 和主板外，还要能够创建内存对象，那么就需要在抽象工厂里面添加创建内存的一个方法。当抽象工厂一发生变化，所有的具体工厂实现都要发生变化，如下、如此就非常的不灵活。

现在有一种相对灵活，但不太安全的改进方式可以解决这个问题，思路如下：抽象工厂里面不需要定义那么多方法，定义一个方法就可以了，给这个方法设置一个参数，通过这个参数来判断具体创建什么产品对象；由于只有一个方法，在返回类型上就不能是具体的某个产品类型了，只能是所有的产品对象都继承或者实现的这么一个类型，比如让所有的产品都实现某个接口，或者干脆使用 Object 类型。

还是通过代码来体会一下，把前面那个示例改造成可扩展的工厂实现。

（1）先来改造抽象工厂。示例代码如下：

```
/**
 * 可扩展的抽象工厂的接口
 */
public interface AbstractFactory {
    /**
```

```
     * 一个通用的创建产品对象的方法，为了简单，直接返回 Object
     * 也可以为所有被创建的产品定义一个公共的接口
     * @param type 具体创建的产品类型标识
     * @return 创建出的产品对象
     */
    public Object createProduct(int type);
}
```

（2）CPU 的接口和实现。主板的接口和实现和前面的示例一样，这里就不再示范了。CPU 分成 Intel 的 CPU 和 AMD 的 CPU，主板分成技嘉的主板和微星的主板。

> **注意** **这里要特别注意传入 createProduct 的参数所代表的含义，这个参数只是用来标识现在是在创建什么类型的产品，比如标识现在是创建 CPU 还是创建主板。一般这个 type 的含义到此就结束了，不再进一步表示具体是什么样的 CPU 或具体是什么样的主板。也就是说 type 不再表示具体是创建 Intel 的 CPU 还是创建 AMD 的 CPU，这就是一个参数所代表的含义的深度问题。要注意，虽然也可以延伸参数的含义到具体的实现上，但这不是可扩展工厂这种设计方式的本意，一般也不这么去做。**

（3）下面来提供具体的工厂实现，也就是相当于以前的装机方案。

先改造原来的方案一吧，现在的实现会有较大的变化。示例代码如下：

```
/**
 * 装机方案一：Intel 的 CPU + 技嘉的主板
 * 这里创建 CPU 和主板对象的时候，是对应的，能匹配上的
 */
public class Schema1 implements AbstractFactory{
    public Object createProduct(int type) {
        Object retObj = null;
        //type 为 1 表示创建 CPU，type 为 2 表示创建主板
        if(type==1){
            retObj = new IntelCPU(1156);
        }else if(type==2){
            retObj = new GAMainboard(1156);
        }
        return retObj;
    }
}
```

用同样的方式来改造原来的方案二。示例代码如下：

```
/**
 * 装机方案二：AMD 的 CPU + 微星的主板
```

```
 * 这里创建 CPU 和主板对象的时候，是对应的，能匹配上的
 */
public class Schema2 implements AbstractFactory{
    public Object createProduct(int type) {
        Object retObj = null;
        //type 为 1 表示创建 CPU，type 为 2 表示创建主板
        if(type==1){
            retObj = new AMDCPU(939);
        }else if(type==2){
            retObj = new MSIMainboard(939);
        }
        return retObj;
    }
}
```

（4）在这个时候使用抽象工厂的客户端实现，也就是在装机工程师类里面，通过抽象工厂来获取相应的配件产品对象。示例代码如下：

```
public  class ComputerEngineer {
    private CPUApi cpu= null;
    private MainboardApi mainboard = null;
    public void makeComputer(AbstractFactory schema){
        prepareHardwares(schema);
    }
    private void prepareHardwares(AbstractFactory schema){
        //这里要去准备 CPU 和主板的具体实现，为了示例简单，这里只准备这两个
        //可是，装机工程师并不知道如何去创建，怎么办

        //使用抽象工厂来获取相应的接口对象
        this.cpu = (CPUApi)schema.createProduct(1);
        this.mainboard = (MainboardApi)schema.createProduct(2);

        //测试一下配件是否好用
        this.cpu.calculate();
        this.mainboard.installCPU();
    }
}
```

通过上面的示例，能看到可扩展工厂的基本实现。从客户端的代码会发现，为什么说这种方式是不太安全的呢？

仔细查看上面蓝色的代码，会发现什么？

你会发现创建产品对象返回来后，需要造型成为具体的对象，因为返回的是 Object，

如果这个时候没有匹配上，比如返回的不是CPU对象，但是要强制造型成为CPU，那么就会发生错误，因此这种实现方式的一个潜在缺点就是不太安全。

（5）下面来体会一下这种方式的灵活性。

假如现在要加入一个新的产品——内存，当然可以提供一个新的装机方案来使用它，这样已有的代码就不需要变化了。

内存接口的示例代码如下：

```
/**
 * 内存的接口
 */
public interface MemoryApi {
    /**
     * 示意方法，内存具有缓存数据的能力
     */
    public void cacheData();
}
```

提供一个现代内存的基本实现。示例代码如下：

```
/**
 * 现代内存的类
 */
public class HyMemory implements MemoryApi{
    public void cacheData() {
        System.out.println("现在正在使用现代内存");
    }
}
```

现在若要使用这个新加入的产品，以前实现的代码都不用变化，只需新添加一个方案，在这个方案里面使用新的产品，然后客户端使用这个新的方案即可。示例代码如下：

```
/**
 * 装机方案三：Intel 的 CPU + 技嘉的主板 + 现代的内存
 */
public class Scheme3 implements AbstractFactory{
    public Object createProduct(int type) {
        Object retObj = null;
        //type 为 1 表示创建 CPU，type 为 2 表示创建主板，type 为 3 表示创建内存
        if(type==1){
            retObj = new IntelCPU(1156);
        }else if(type==2){
            retObj = new GAMainboard(1156);
        }
```

```
        //创建新添加的产品
        else if(type==3){
            retObj = new HyMemory();
        }
        return retObj;
    }
}
```

这个时候的装机工程师类，如果要创建带内存的电脑，需要在装机工程师类里面添加对内存的使用。示例代码如下：

```
public  class ComputerEngineer {
    private CPUApi cpu= null;
    private MainboardApi mainboard = null;
    /**
     * 定义组装电脑需要的内存
     */
    private MemoryApi memory = null;
    public void makeComputer(AbstractFactory schema){
        prepareHardwares(schema);
    }
    private void prepareHardwares(AbstractFactory schema){
        //使用抽象工厂来获取相应的接口对象
        this.cpu = (CPUApi)schema.createProduct(1);
        this.mainboard = (MainboardApi)schema.createProduct(2);
        this.memory = (MemoryApi)schema.createProduct(3);

        //测试一下配件是否好用
        this.cpu.calculate();
        this.mainboard.installCPU();
        if(memory!=null){
            this.memory.cacheData();
        }
    }
}
```

可能有朋友会发现上面蓝色的代码中内存操作的地方，跟前面 CPU 和主板的操作方式不一样，多了一个 if 判断。原因是为了要同时满足以前和现在的要求，如果是以前的客户端，它调用的时候就没有内存，这个时候操作内存就会出错，因此添加一个判断，有内存的时候才操作内存，就不会出错了。

此时的客户端，只要选择使用方案三就可以了。示例代码如下：

```
public class Client {
```

```
    public static void main(String[] args) {
        ComputerEngineer engineer = new ComputerEngineer();
        AbstractFactory schema = new Schema3();
        engineer.makeComputer(schema);
    }
}
```

运行结果如下：

```
now in Intel CPU,pins=1156
now in GAMainboard,cpuHoles=1156
现在正在使用现代内存
```

测试一下看看，体会一下这种设计方式的灵活性。当然前面也讲到了，这种方式可能会不太安全，至于是否使用，就看具体应用设计上的权衡了。

7.3.3　抽象工厂模式和 DAO

1. 什么是 DAO

DAO：数据访问对象，是 Data Access Object 首字母的简写。

DAO 是 JEE（也称 JavaEE，原 J2EE）中的一个标准模式，通过它来解决访问数据对象所面临的一系列问题，比如，数据源不同、存储类型不同、访问方式不同、供应商不同、版本不同等，这些不同会造成访问数据的实现上差别很大。

- 数据源的不同，比如存放于数据库的数据源，存放于 LDAP（轻型目录访问协议）的数据源；又比如存放于本地的数据源和远程服务器上的数据源等。
- 存储类型的不同，比如关系型数据库（RDBMS）、面向对象数据库（ODBMS）、纯文件、XML 等。
- 访问方式的不同，比如访问关系型数据库，可以用 JDBC、EntityBean、JPA 等来实现，当然也可以采用一些流行的框架，如 Hibernate、IBatis 等。
- 供应商的不同，比如关系型数据库，流行的如 Oracel、DB2、SqlServer、MySQL 等等，它们的供应商是不同的。
- 版本不同，比如关系型数据库，不同的版本，实现的功能是有差异的，就算是对标准的 SQL 的支持，也是有差异的。

但是对于需要进行数据访问的逻辑层而言，它可不想面对这么多不同，也不想处理这么多差异，它希望能以一个统一的方式来访问数据。

此时系统结构如图 7.4 所示。

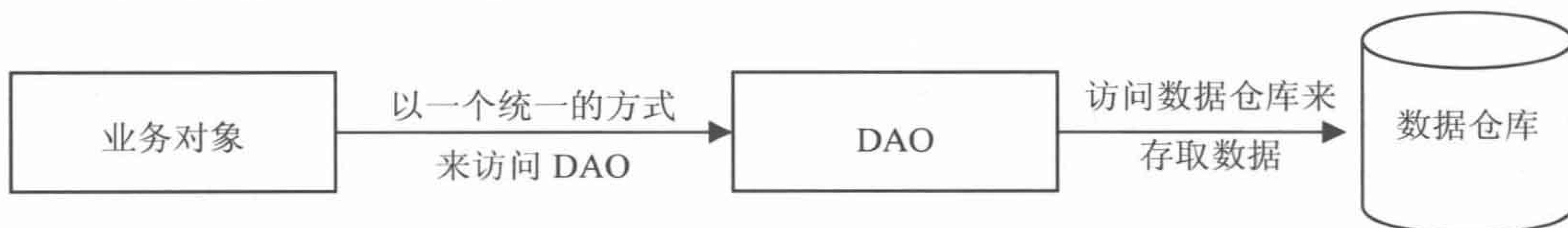


图 7.4　业务对象访问 DAO 的示意图

也就是说，DAO 需要抽象和封装所有对数据的访问，DAO 承担和数据仓库交互的职责，这也意味着，访问数据所面临的所有问题，都需要 DAO 在内部来自行解决。

2. DAO 和抽象工厂的关系

了解了什么是 DAO 后，可能有些朋友会想，DAO 同抽象工厂模式有什么关系呢？看起来好像是完全不靠边啊。

事实上，在实现 DAO 模式的时候，最常见的实现策略就是使用工厂的策略，而且多是通过抽象工厂模式来实现，当然在使用抽象工厂模式来实现的时候，可以结合工厂方法模式。因此 DAO 模式和抽象工厂模式有很大的联系。

3. DAO 模式的工厂实现策略

下面就来看看 DAO 模式实现的时候是如何采用工厂方法和抽象工厂的。

1）采用工厂方法模式

假如现在在一个订单处理的模块里面。大家都知道，订单通常分成两个部分，一部分是订单主记录或者是订单主表，另一部分是订单明细记录或者是订单子表，那么现在业务对象需要操作订单的主记录，也需要操作订单的子记录。

如果这个时候的业务比较简单，而且对数据的操作是固定的，比如就是操作数据库，不管订单的业务如何变化，底层数据存储都是一样的，那么这种情况下，可以采用工厂方法模式，此时系统结构如图 7.5 所示。

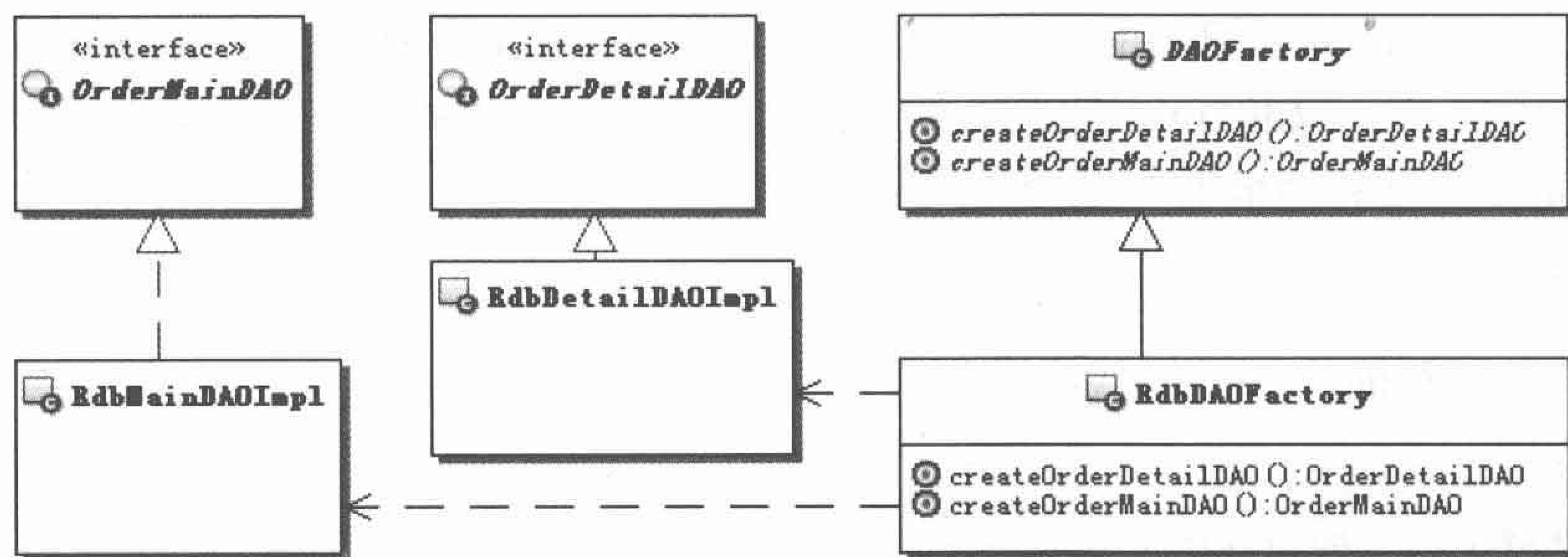


图 7.5　DAO 模式的工厂方法实现策略结构示意图

从上面的结构示意图可以看出，如果底层存储固定的时候，DAOFactory 就相当于工厂方法模式中的 Creator，在里面定义两个工厂方法，分别创建订单主记录的 DAO 对象和创建订单子记录的 DAO 对象，因为固定是数据库实现，因此提供一个具体的工厂 RdbDAOFactory（Rdb，关系型数据库）来实现对象的创建。也就是说 DAO 可以采用工厂方法模式来实现。

采用工厂方法模式的情况，要求 DAO 底层存储实现方式是固定的，这种模式多用在一些简单的小项目的开发上。

2）采用抽象工厂模式

实际上更多的时候 DAO 底层存储实现方式是**不固定**的，DAO 通常会支持多种存储实现方式，具体使用哪一种存储方式可能是由应用动态决定，或者是通过配置来指定。这种情况多见于产品开发，或者是稍复杂的应用、亦或较大的项目中。

对于底层存储方式不固定的时候，一般采用抽象工厂模式来实现 DAO。比如现在的

实现除了 RDB 的实现，还会有 Xml 的实现，它们会被应用动态的选择，此时系统结构如图 7.6 所示。

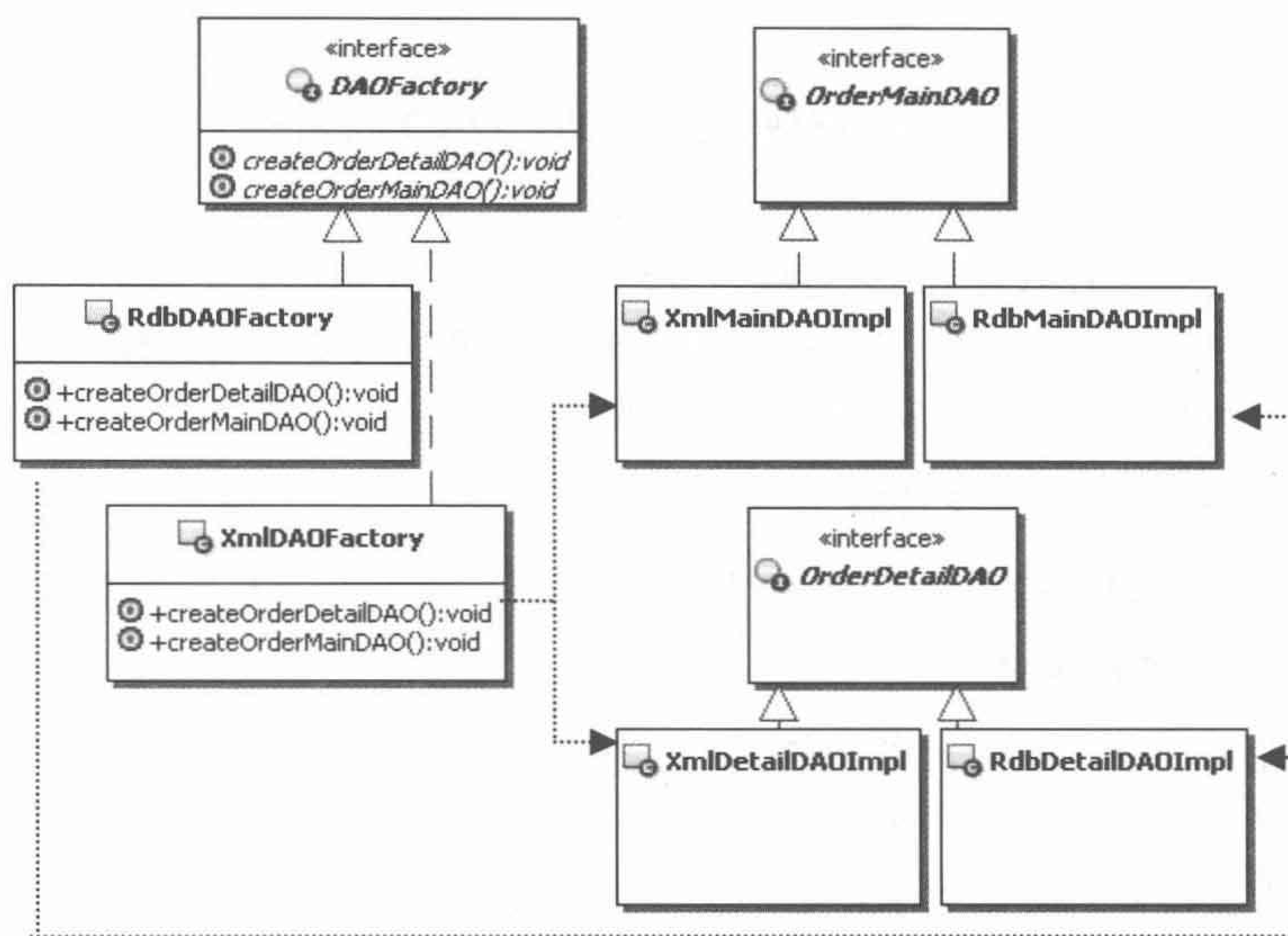


图 7.6　DAO 模式的抽象工厂实现策略结构示意图

从上面的结构示意图可以看出，采用抽象工厂模式来实现 DAO 的时候，DAOFactory 就相当于抽象工厂，里面定义一系列创建相关对象的方法，分别是创建订单主记录的 DAO 对象和创建订单子记录的 DAO 对象，此时 OrderMainDAO 和 OrderDetailDAO 就相当于被创建的产品，RdbDAOFactory 和 XmlDAOFactory 就相当于抽象工厂的具体实现，在它们里面会选择相应的具体的产品实现来创建对象。

4. 代码示例使用抽象工厂实现 DAO 模式

（1）先看看抽象工厂的代码实现。示例代码如下：

```
/**
 * 抽象工厂，创建订单主、子记录对应的 DAO 对象
 */
public abstract class DAOFactory {
    /**
     * 创建订单主记录对应的 DAO 对象
     * @return 订单主记录对应的 DAO 对象
     */
    public abstract OrderMainDAO createOrderMainDAO();
    /**
     * 创建订单子记录对应的 DAO 对象
     * @return 订单子记录对应的 DAO 对象
     */
    public abstract OrderDetailDAO createOrderDetailDAO();
}
```

（2）看看产品对象的接口，就是订单主、子记录的 DAO 定义。

先来看看订单主记录的 DAO 定义。示例代码如下：

```
/**
 * 订单主记录对应的 DAO 操作接口
 */
public interface OrderMainDAO {
    /**
     * 示意方法，保存订单主记录
     */
    public void saveOrderMain();
}
```

再来看看订单子记录的 DAO 定义。示例代码如下：

```
/**
 * 订单子记录对应的 DAO 操作接口
 */
public interface OrderDetailDAO {
    /**
     * 示意方法，保存订单子记录
     */
    public void saveOrderDetail();
}
```

（3）接下来实现订单主、子记录的 DAO。

先来看看关系型数据库的实现方式，示例代码如下：

```
public class RdbMainDAOImpl implements OrderMainDAO{
    public void saveOrderMain() {
        System.out.println("now in RdbMainDAOImpl saveOrderMain");
    }
}
public class RdbDetailDAOImpl implements OrderDetailDAO{
    public void saveOrderDetail() {
        System.out.println(
                        "now in RdbDetailDAOImpl saveOrderDetail");
    }
}
```

Xml 实现的方式一样。为了演示简单，都是输出了一句话。示例代码如下：

```
public class XmlMainDAOImpl implements OrderMainDAO{
    public void saveOrderMain() {
        System.out.println("now in XmlMainDAOImpl saveOrderMain");
    }
```

```
}
public class XmlDetailDAOImpl implements OrderDetailDAO{
    public void saveOrderDetail() {
        System.out.println("now in XmlDAOImpl2 saveOrderDetail");
    }
}
```

（4）再看看具体的工厂实现。

先来看看关系型数据库实现方式的工厂。示例代码如下：

```
public class RdbDAOFactory extends DAOFactory{
    public OrderDetailDAO createOrderDetailDAO() {
        return new RdbDetailDAOImpl();
    }
    public OrderMainDAO createOrderMainDAO() {
        return new RdbMainDAOImpl();
    }
}
```

Xml 实现方式的工厂的示例代码如下：

```
public class XmlDAOFactory extends DAOFactory {
    public OrderDetailDAO createOrderDetailDAO() {
        return new XmlDetailDAOImpl();
    }
    public OrderMainDAO createOrderMainDAO() {
        return new XmlMainDAOImpl();
    }
}
```

（5）好了，使用抽象工厂简单地实现了 DAO 模式。在客户端通常是由业务对象来调用 DAO，那么该怎么使用这个 DAO 呢？示例代码如下：

```
public class BusinessObject {
    public static void main(String[] args) {
        //创建 DAO 的抽象工厂
        DAOFactory df = new RdbDAOFactory();
        //通过抽象工厂来获取需要的 DAO 接口
        OrderMainDAO mainDAO = df.createOrderMainDAO();
        OrderDetailDAO detailDAO = df.createOrderDetailDAO();
        //调用 DAO 来完成数据存储的功能
        mainDAO.saveOrderMain();
        detailDAO.saveOrderDetail();
    }
}
```

通过上面的示例，可以看出 DAO 可以采用抽象工厂模式来实现，这也是大部分 DAO 实现所采用的方式。

7.3.4 抽象工厂模式的优缺点

抽象工厂模式的优点

- 分离接口和实现

  客户端使用抽象工厂来创建需要的对象，而客户端根本就不知道具体的实现是谁，客户端只是面向产品的接口编程而已。也就是说，客户端从具体的产品实现中解耦。

- 使得切换产品簇变得容易

  因为一个具体的工厂实现代表的是一个产品簇，比如上面例子的 Scheme1 代表装机方案一：Intel 的 CPU + 技嘉的主板，如果要切换成为 Scheme2，那就变成了装机方案二：AMD 的 CPU + 微星的主板。

  客户端选用不同的工厂实现，就相当于是在切换不同的产品簇。

抽象工厂模式的缺点

- 不太容易扩展新的产品

  前面也提到这个问题了，如果需要给整个产品簇添加一个新的产品，那么就需要修改抽象工厂，这样就会导致修改所有的工厂实现类。在前面提供了一个可以扩展工厂的方式来解决这个问题，但是又不够安全。如何选择，则要根据实际应用来权衡。

- 容易造成类层次复杂

  在使用抽象工厂模式的时候，如果需要选择的层次过多，那么会造成整个类层次变得复杂。

  举个例子来说，就比如前面讲到的 DAO 的示例，现在这个 DAO 只有一个选择的层次，也就是选择是使用关系型数据库来实现，还是用 Xml 来实现。现在考虑这样一种情况，如果关系型数据库实现里面又分成几种，比如，基于 Oracle 的实现、基于 SqlServer 的实现、基于 MySql 的实现等。

  那么客户端怎么选择呢？不会把所有可能的实现情况全部都做到一个层次上吧，这个时候客户端就需要一层一层地选择，也就是整个**抽象工厂的实现也需要分出层次来，每一层负责一种选择，也就是一层屏蔽一种变化，**这样很容易造成复杂的类层次结构。

7.3.5 思考抽象工厂模式

> 抽象工厂模式的本质：**选择产品簇的实现。**

1. 抽象工厂模式的本质

工厂方法是选择单个产品的实现，虽然一个类里面可以有多个工厂方法，但是这些方法之间一般是没有联系的，即使看起来像有联系。

但是抽象工厂着重的就是为一个产品簇选择实现，定义在抽象工厂里面的方法通常是有联系的，它们都是产品的某一部分或者是相互依赖的。如果抽象工厂里面只定义一个方法，直接创建产品，那么就退化成为工厂方法了。

2. 何时选用抽象工厂模式

建议在以下情况中选用抽象工厂模式。

- 如果希望一个系统独立于它的产品的创建、组合和表示的时候。换句话说，希望一个系统只是知道产品的接口，而不关心实现的时候。
- 如果一个系统要由多个产品系列中的一个来配置的时候。换句话说，就是可以动态地切换产品簇的时候。
- 如果要强调一系列相关产品的接口，以便联合使用它们的时候。

7.3.6 相关模式

- 抽象工厂模式和工厂方法模式

  这两个模式既有区别，又有联系，可以组合使用。

  工厂方法模式一般是针对单独的产品对象的创建，而抽象工厂模式注重产品簇对象的创建，这是它们的区别。

  如果把抽象工厂创建的产品簇简化，这个产品簇就只有一个产品，那么这个时候的抽象工厂跟工厂方法是差不多的，也就是抽象工厂可以退化成工厂方法，而工厂方法又可以退化成简单工厂，这也是它们的联系。

  在抽象工厂的实现中，还可以使用工厂方法来提供抽象工厂的具体实现，也就是说它们可以组合使用。

- 抽象工厂模式和单例模式

  这两个模式可以组合使用。

  在抽象工厂模式里面，具体的工厂实现，在整个应用中，通常一个产品系列只需要一个实例就可以了，因此可以把具体的工厂实现成为单例。

第 8 章　生成器模式（Builder）

研磨

8.1 场景问题

8.1.1 继续导出数据的应用框架

在讨论工厂方法模式的时候，提到了一个导出数据的应用框架。

对于导出数据的应用框架，通常在导出数据上，会有一些约定的方式，比如导出成文本格式、数据库备份形式、Excel 格式、Xml 格式等。

在工厂方法模式章节里面，讨论并使用工厂方法模式来解决了如何选择具体导出方式的问题，并没有涉及到每种方式具体如何实现。

换句话说，在讨论工厂方法模式的时候，并没有讨论如何实现导出成文本、Xml 等具体的格式，本章就来讨论这个问题。

对于导出数据的应用框架，通常对于具体的导出内容和格式是有要求的，假如现在有如下的要求，简单描述一下：

- 导出的文件，不管什么格式，都分成 3 个部分，分别是文件头、文件体和文件尾。
- 在文件头部分，需要描述如下信息：分公司或门市点编号、导出数据的日期，对于文本格式，中间用逗号分隔。
- 在文件体部分，需要描述如下信息：表名称，然后分条描述数据。对于文本格式，表名称单独占一行，数据描述一行算一条数据，字段间用逗号分隔。
- 在文件尾部分，需要描述如下信息：输出人。

现在就要来实现上述功能。为了演示简单点，在工厂方法模式里面已经实现的功能，就再不重复了，这里只关心如何实现导出文件，而且只实现导出成文本格式和 Xml 格式就可以了，其他就不用考虑了。

8.1.2 不用模式的解决方案

不就是要实现导出数据到文本文件和 XML 文件吗？其实不管什么格式，需要导出的数据是一样的，只是具体导出到文件中的内容则会随着格式的不同而不同。

（1）下面将描述文件各个部分的数据对象定义出来。

先来看看描述输出到文件头的内容的对象。示例代码如下：

```
/**
 * 描述输出到文件头的内容的对象
 */
public class ExportHeaderModel {
    /**
    * 分公司或门市点编号
     */
    private String depId;
    /**
```

```
     * 导出数据的日期
     */
    private String exportDate;
    public String getDepId() {
        return depId;
    }
    public void setDepId(String depId) {
        this.depId = depId;
    }
    public String getExportDate() {
        return exportDate;
    }
    public void setExportDate(String exportDate) {
        this.exportDate = exportDate;
    }
}
```

接下来看看描述输出数据的对象。示例代码如下：

```
/**
 * 描述输出数据的对象
 */
public class ExportDataModel {
    /**
     * 产品编号
     */
    private String productId;
    /**
     * 销售价格
     */
    private double price;
    /**
     * 销售数量
     */
    private double amount;

    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
```

```
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```

再来看看描述输出到文件尾的内容的对象。示例代码如下：

```
/**
 * 描述输出到文件尾的内容的对象
 */
public class ExportFooterModel {
    /**
     * 输出人
     */
    private String exportUser;
    public String getExportUser() {
        return exportUser;
    }
    public void setExportUser(String exportUser) {
        this.exportUser = exportUser;
    }
}
```

（2）下面来具体地看看导出的实现。

先看看导出数据到文本文件的对象，主要就是要实现拼接输出的内容。示例代码如下：

```
/**
 * 导出数据到文本文件的对象
 */
public class ExportToTxt {
    /**
     * 导出数据到文本文件
     * @param ehm 文件头的内容
```

```
     * @param mapData 数据的内容
     * @param efm 文件尾的内容
     */
    public void export(ExportHeaderModel ehm
                ,Map<String,Collection<ExportDataModel>> mapData
                ,ExportFooterModel efm){
        //用来记录最终输出的文件内容
        StringBuffer buffer = new StringBuffer();
        //1: 先来拼接文件头的内容
        buffer.append(ehm.getDepId()+","
                                +ehm.getExportDate()+"\n");
        //2: 接着来拼接文件体的内容
        for(String tblName : mapData.keySet()){
            //先拼接表名称
            buffer.append(tblName+"\n");
            //然后循环拼接具体数据
            for(ExportDataModel edm : mapData.get(tblName)){
                buffer.append(edm.getProductId()+","
                        +edm.getPrice()+","+edm.getAmount()+"\n");
            }
        }
        //3: 接着来拼接文件尾的内容
        buffer.append(efm.getExportUser());

        //为了演示的简洁性，这里就不再写输出文件的代码了
        //把要输出的内容输出到控制台看看
        System.out.println("输出到文本文件的内容: \n"+buffer);
    }
}
```

（3）接下来看看导出数据到 XML 文件的对象，比较麻烦，要按照 XML 的格式进行拼接。示例代码如下：

```
/**
 * 导出数据到XML文件的对象
 */
public class ExportToXml {
    /**
     * 导出数据到XML文件
     * @param ehm 文件头的内容
     * @param mapData 数据的内容
     * @param efm 文件尾的内容
```

```
    */
    public void export(ExportHeaderModel ehm
            ,Map<String,Collection<ExportDataModel>> mapData
            ,ExportFooterModel efm){
        //用来记录最终输出的文件内容
        StringBuffer buffer = new StringBuffer();
        //1：先来拼接文件头的内容
        buffer.append(
                "<?xml version='1.0' encoding='gb2312'?>\n");
        buffer.append("<Report>\n");
        buffer.append("  <Header>\n");
        buffer.append("    <DepId>"+ehm.getDepId()+"</DepId>\n");
        buffer.append("    <ExportDate>"+ehm.getExportDate()
                                +"</ExportDate>\n");
        buffer.append("  </Header>\n");
        //2：再来拼接文件体的内容
        buffer.append("  <Body>\n");
        for(String tblName : mapData.keySet()){
            //先拼接表名称
            buffer.append(
                    "    <Datas TableName=\""+tblName+"\">\n");
            //然后循环拼接具体数据
            for(ExportDataModel edm : mapData.get(tblName)){
                buffer.append("      <Data>\n");
                buffer.append("        <ProductId>"
                        +edm.getProductId()+"</ProductId>\n");
                buffer.append("        <Price>"+edm.getPrice()
                        +"</Price>\n");
                buffer.append("        <Amount>"+edm.getAmount()
                        +"</Amount>\n");
                buffer.append("      </Data>\n");
            }
            buffer.append("    </Datas>\n");
        }
        buffer.append("  </Body>\n");
        //3：接着来拼接文件尾的内容
        buffer.append("  <Footer>\n");
        buffer.append("    <ExportUser>"+efm.getExportUser()
                +"</ExportUser>\n");
        buffer.append("  </Footer>\n");
```

```
        buffer.append("</Report>\n");

        //为了演示的简洁性，这里就不再写输出文件的代码了
        //把要输出的内容输出到控制台看看
        System.out.println("输出到XML文件的内容：\n"+buffer);
    }
}
```

（4）看看客户端，如何来使用这些对象。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备测试数据
        ExportHeaderModel ehm = new ExportHeaderModel();
        ehm.setDepId("一分公司");
        ehm.setExportDate("2010-05-18");

        Map<String,Collection<ExportDataModel>> mapData =
                new HashMap<String,Collection<ExportDataModel>>();
        Collection<ExportDataModel> col =
                new ArrayList<ExportDataModel>();

        ExportDataModel edm1 = new ExportDataModel();
        edm1.setProductId("产品001号");
        edm1.setPrice(100);
        edm1.setAmount(80);

        ExportDataModel edm2 = new ExportDataModel();
        edm2.setProductId("产品002号");
        edm2.setPrice(99);
        edm2.setAmount(55);
        //把数据组装起来
        col.add(edm1);
        col.add(edm2);
        mapData.put("销售记录表", col);

        ExportFooterModel efm = new ExportFooterModel();
        efm.setExportUser("张三");
        //测试输出到文本文件
        ExportToTxt toTxt = new ExportToTxt();
        toTxt.export(ehm, mapData, efm);
        //测试输出到xml文件
```

```
        ExportToXml toXml = new ExportToXml();
        toXml.export(ehm, mapData, efm);
    }
}
```

运行结果如下：

```
输出到文本文件的内容：
一分公司,2010-05-18
销售记录表
产品001号,100.0,80.0
产品002号,99.0,55.0
张三
输出到XML文件的内容：
<?xml version='1.0' encoding='gb2312'?>
<Report>
  <Header>
    <DepId>一分公司</DepId>
    <ExportDate>2010-05-18</ExportDate>
  </Header>
  <Body>
    <Datas TableName="销售记录表">
      <Data>
        <ProductId>产品001号</ProductId>
        <Price>100.0</Price>
        <Amount>80.0</Amount>
      </Data>
      <Data>
        <ProductId>产品002号</ProductId>
        <Price>99.0</Price>
        <Amount>55.0</Amount>
      </Data>
    </Datas>
  </Body>
  <Footer>
    <ExportUser>张三</ExportUser>
  </Footer>
</Report>
```

8.1.3　有何问题

仔细观察上面的实现，会发现，不管是输出成文本文件，还是输出到 XML 文件，在实现的时候，步骤基本上都是一样的，大致分成了以下四步。

（1）先拼接文件头的内容。

（2）然后拼接文件体的内容。

（3）再拼接文件尾的内容。

（4）最后把拼接好的内容输出去成为文件。

也就是说，对于不同的输出格式，处理步骤是一样的，但是每步的具体实现是不一样的。按照现在的实现方式，就存在如下的问题。

（1）构建每种输出格式的文件内容的时候，都会重复这几个处理步骤，应该提炼出来，形成公共的处理过程。

（2）今后可能会有很多不同输出格式的要求，这就需要在处理过程不变的情况下，能方便地切换不同的输出格式的处理。

换句话来说，也就是构建每种格式的数据文件的处理过程，应该和具体的步骤实现分开，这样就能够复用处理过程，而且能很容易地切换不同地输出格式。

可是该如何实现呢？

8.2　解 决 方 案

8.2.1　使用生成器模式来解决问题

用来解决上述问题的一个合理的解决方案就是生成器模式。那么什么是生成器模式呢？

1. 生成器模式的定义

> 将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

2. 应用生成器模式来解决问题的思路

仔细分析上面的实现，构建每种格式的数据文件的处理过程，这不就是构建过程吗？而每种格式具体的步骤实现，不就相当于是不同的表示吗？因为不同的步骤实现，决定了最终的表现也就不同。也就是说，上面的问题恰好就是生成器模式要解决的问题。

要实现同样的构建过程可以创建不同的表现，那么一个自然的思路就是先把构建过程独立出来，在生成器模式中把它称为指导者，由它来指导装配过程，但是不负责每步具体的实现。当然，光有指导者是不够的，必须要有能具体实现每步的对象，在生成器

模式中称这些实现对象为生成器。

这样一来，指导者就是可以重用的构建过程，而生成器是可以被切换的具体实现。前面的实现中，每种具体的导出文件格式的实现就相当于生成器。

8.2.2 生成器模式的结构和说明

生成器模式的结构如图 8.1 所示。

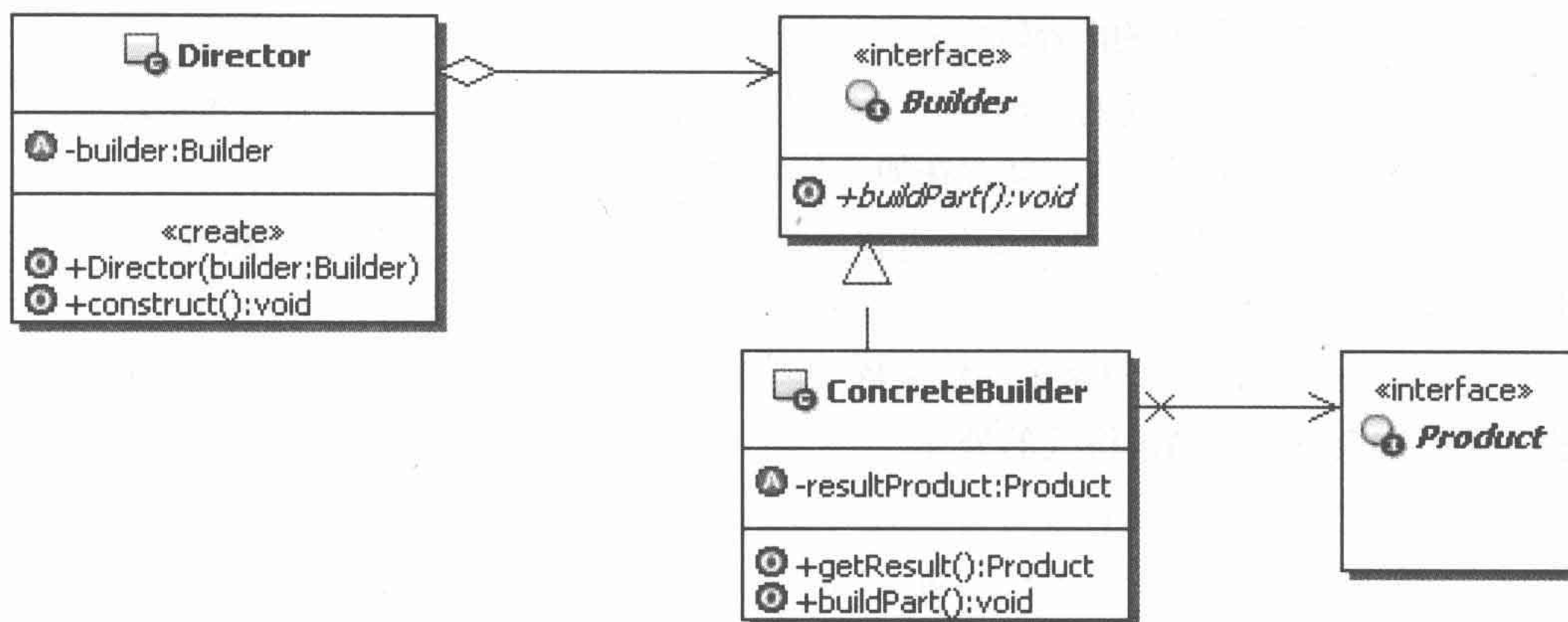


图 8.1 生成器模式结构示意图

- Builder：生成器接口，定义创建一个 Product 对象所需的各个部件的操作。
- ConcreteBuilder：具体的生成器实现，实现各个部件的创建，并负责组装 Product 对象的各个部件，同时还提供一个让用户获取组装完成后的产品对象的方法。
- Director：指导者，也被称为导向者，主要用来使用 Builder 接口，以一个统一的过程来构建所需要的 Product 对象。
- Product：产品，表示被生成器构建的复杂对象，包含多个部件。

8.2.3 生成器模式示例代码

（1）生成器接口定义的示例代码如下：

```
/**
 * 生成器接口，定义创建一个产品对象所需的各个部件的操作
 */
public interface Builder {
    /**
     * 示意方法，构建某个部件
     */
    public void buildPart();
}
```

（2）具体生成器实现的示例代码如下：

```
/**
 * 具体的生成器实现对象
 */
public class ConcreteBuilder implements Builder {
    /**
     * 生成器最终构建的产品对象
     */
    private Product resultProduct;
    /**
     * 获取生成器最终构建的产品对象
     * @return 生成器最终构建的产品对象
     */
    public Product getResult() {
        return resultProduct;
    }

    public void buildPart() {
        //构建某个部件的功能处理
    }
}
```

（3）相应的产品对象接口的示例代码如下：

```
/**
 * 被构建的产品对象的接口
 */
public interface Product {
    //定义产品的操作
}
```

（4）最后来看看指导者的实现示意。示例代码如下：

```
/**
 * 指导者，指导使用生成器的接口来构建产品的对象
 */
public class Director {
    /**
     * 持有当前需要使用的生成器对象
     */
    private Builder builder;

    /**
     * 构造方法，传入生成器对象
```

```
     * @param builder 生成器对象
     */
    public Director(Builder builder) {
        this.builder = builder;
    }

    /**
     * 示意方法，指导生成器构建最终的产品对象
     */
    public void construct() {
        //通过使用生成器接口来构建最终的产品对象
        builder.buildPart();
    }
}
```

8.2.4 使用生成器模式重写示例

要使用生成器模式来重写示例，重要的任务就是要把指导者和生成器接口定义出来。指导者就是用来执行那四个步骤的对象，而生成器是用来实现每种格式下，对于每个步骤的具体实现的对象。

按照生成器模式重写示例的结构如图 8.2 所示。

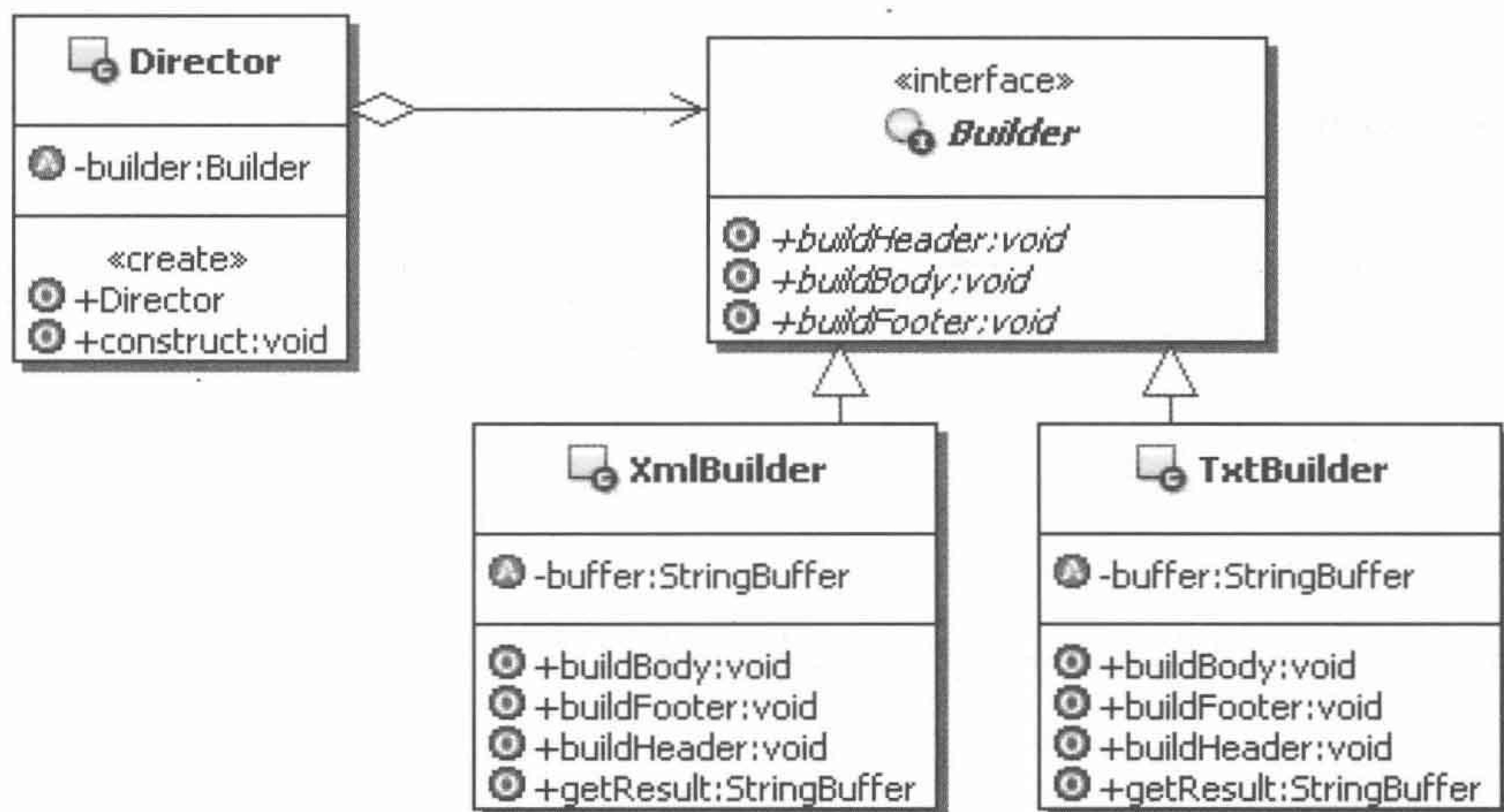


图 8.2 生成器模式重写示例的结构示意图

下面还是一起来看看代码，会比较清楚。

（1）前面示例中的三个数据模型对象还继续沿用，这里就不再赘述了。

（2）下面来看看定义的 Builder 接口，主要是把导出各种格式文件的处理过程的步骤定义出来，每个步骤负责构建最终导出文件的一部分。示例代码如下：

```
/**
 * 生成器接口，定义创建一个输出文件对象所需的各个部件的操作
 */
public interface Builder {
    /**
     * 构建输出文件的Header部分
     * @param ehm 文件头的内容
     */
    public void buildHeader(ExportHeaderModel ehm);
    /**
     * 构建输出文件的Body部分
     * @param mapData 要输出的数据的内容
     */
    public void buildBody(
                Map<String,Collection<ExportDataModel>> mapData);
    /**
     * 构建输出文件的Footer部分
     * @param efm 文件尾的内容
     */
    public void buildFooter(ExportFooterModel efm);
}
```

（3）接下来看看具体的生成器实现。其实就是把原来示例中写在一起的实现，拆分成多个步骤实现。

先来看导出数据到文本文件的生成器实现。示例代码如下：

```
/**
 * 实现导出数据到文本文件的生成器对象
 */
public class TxtBuilder implements Builder {
    /**
     * 用来记录构建的文件的内容，相当于产品
     */
    private StringBuffer buffer = new StringBuffer();

    public void buildBody(
              Map<String, Collection<ExportDataModel>> mapData) {
        for(String tblName : mapData.keySet()){
            //先拼接表名称
            buffer.append(tblName+"\n");
            //然后循环拼接具体数据
```

```
            for(ExportDataModel edm : mapData.get(tblName)){
                buffer.append(edm.getProductId()+","
                        +edm.getPrice()+","+edm.getAmount()+"\n");
            }
        }
    }
    public void buildFooter(ExportFooterModel efm) {
        buffer.append(efm.getExportUser());
    }
    public void buildHeader(ExportHeaderModel ehm) {
        buffer.append(ehm.getDepId()+","
                                +ehm.getExportDate()+"\n");
    }
    public StringBuffer getResult(){
        return buffer;
    }
}
```

再看看导出数据到 XML 文件的生成器实现。示例代码如下：

```
/**
 * 实现导出数据到XML文件的生成器对象
 */
public class XmlBuilder implements Builder {
    /**
     * 用来记录构建的文件的内容，相当于产品
     */
    private StringBuffer buffer = new StringBuffer();

    public void buildBody(
                Map<String, Collection<ExportDataModel>> mapData){
        buffer.append("  <Body>\n");
        for(String tblName : mapData.keySet()){
            //先拼接表名称
            buffer.append(
                        "    <Datas TableName=\""+tblName+"\">\n");
            //然后循环拼接具体数据
            for(ExportDataModel edm : mapData.get(tblName)){
                buffer.append("      <Data>\n");
                buffer.append("        <ProductId>"
                        +edm.getProductId()+"</ProductId>\n");
                buffer.append("        <Price>"+edm.getPrice()
```

```
                                                    +"</Price>\n");
                buffer.append("       <Amount>"+edm.getAmount()
                                                    +"</Amount>\n");
                buffer.append("     </Data>\n");
            }
            buffer.append("   </Datas>\n");
        }
        buffer.append("  </Body>\n");
    }
    public void buildFooter(ExportFooterModel efm) {
        buffer.append("  <Footer>\n");
        buffer.append("    <ExportUser>"+efm.getExportUser()
+"</ExportUser>\n");
        buffer.append("  </Footer>\n");
        buffer.append("</Report>\n");
    }
    public void buildHeader(ExportHeaderModel ehm) {
        buffer.append(
                    "<?xml version='1.0' encoding='gb2312'?>\n");
        buffer.append("<Report>\n");
        buffer.append("  <Header>\n");
        buffer.append("    <DepId>"+ehm.getDepId()+"</DepId>\n");
        buffer.append("    <ExportDate>"+ehm.getExportDate()
                                                +"</ExportDate>\n");
        buffer.append("  </Header>\n");
    }
    public StringBuffer getResult(){
        return buffer;
    }
}
```

（4）指导者。有了具体的生成器实现后，需要由指导者来指导它进行具体的产品构建。由于构建的产品是文本内容，所以就不用单独定义产品对象了。示例代码如下：

```
/**
 * 指导者，指导使用生成器的接口来构建输出的文件的对象
 */
public class Director {
    /**
     * 持有当前需要使用的生成器对象
```

```
     */
    private Builder builder;
    /**
     * 构造方法，传入生成器对象
     * @param builder 生成器对象
     */
    public Director(Builder builder) {
        this.builder = builder;
    }

    /**
     * 指导生成器构建最终的输出的文件的对象
     * @param ehm 文件头的内容
     * @param mapData 数据的内容
     * @param efm 文件尾的内容
     */
    public void construct(ExportHeaderModel ehm,
                Map<String,Collection<ExportDataModel>> mapData,
                ExportFooterModel efm) {
        //1：先构建Header
        builder.buildHeader(ehm);
        //2：然后构建Body
        builder.buildBody(mapData);
        //3：再构建Footer
        builder.buildFooter(efm);
    }
}
```

（5）都实现得差不多了，下面编写客户端程序测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备测试数据
        ExportHeaderModel ehm = new ExportHeaderModel();
        ehm.setDepId("一分公司");
        ehm.setExportDate("2010-05-18");
```

```
Map<String,Collection<ExportDataModel>> mapData =
        new HashMap<String,Collection<ExportDataModel>>();
Collection<ExportDataModel> col =
                        new ArrayList<ExportDataModel>();

ExportDataModel edm1 = new ExportDataModel();
edm1.setProductId("产品 001 号");
edm1.setPrice(100);
edm1.setAmount(80);

ExportDataModel edm2 = new ExportDataModel();
edm2.setProductId("产品 002 号");
edm2.setPrice(99);
edm2.setAmount(55);
//把数据组装起来
col.add(edm1);
col.add(edm2);
mapData.put("销售记录表", col);

ExportFooterModel efm = new ExportFooterModel();
efm.setExportUser("张三");

//测试输出到文本文件
TxtBuilder txtBuilder = new TxtBuilder();
//创建指导者对象
Director director = new Director(txtBuilder);
director.construct(ehm, mapData, efm);
//把要输出的内容输出到控制台看看
System.out.println("输出到文本文件的内容：\n"
                              +txtBuilder.getResult());
//测试输出到 XML 文件
XmlBuilder xmlBuilder = new XmlBuilder();
Director director2 = new Director(xmlBuilder);
```

```
        director2.construct(ehm, mapData, efm);
        //把要输出的内容输出到控制台看看
        System.out.println("输出到XML文件的内容：\n"
                                            +xmlBuilder.getResult());
    }
}
```

看了上面的示例会发现，其实生成器模式也挺简单的，好好体会一下。通过上面的讲述，应该能很清晰地看出生成器模式的实现方式和它的优势所在了，那就是对同一个构建过程，只要配置不同的生成器实现，就会生成不同表现的对象。

8.3 模式讲解

8.3.1 认识生成器模式

1. 生成器模式的功能

生成器模式的主要功能是构建复杂的产品，而且是细化的、分步骤的构建产品，也就是生成器模式重在一步一步解决构造复杂对象的问题。如果仅仅这么认识生成器模式的功能是不够的。

> 提示 **更为重要的是，这个构建的过程是统一的、固定不变的，变化的部分放到生成器部分了，只要配置不同的生成器，那么同样的构建过程，就能构建出不同的产品来。**

再直白点说，生成器模式的重心在于分离构建算法和具体的构造实现，从而使得构建算法可以重用。具体的构造实现可以很方便地扩展和切换，从而可以灵活地组合来构造出不同的产品对象。

2. 生成器模式的构成

要特别注意，生成器模式分成两个很重要的部分。

- 一个部分是Builder接口，这里是定义了如何构建各个部件，也就是知道每个部件功能如何实现，以及如何装配这些部件到产品中去；
- 另外一个部分是Director，Director是知道如何组合来构建产品，也就是说Director负责整体的构建算法，而且通常是分步骤地来执行。

不管如何变化，**Builder模式都存在这么两个部分，一个部分是部件构造和产品装配，另一个部分是整体构建的算法**。认识这点是很重要的，因为在生成器模式中，强调的是固定整体构建的算法，而灵活扩展和切换部件的具体构造和产品装配的方式，所以要严格区分这两个部分。

在Director实现整体构建算法的时候，遇到需要创建和组合具体部件的时候，就会

把这些功能通过委托，交给 Builder 去完成。

3. 生成器模式的使用

应用生成器模式的时候，可以让客户端创造 Director，在 Director 里面封装整体构建算法，然后让 Director 去调用 Builder，让 Builder 来封装具体部件的构建功能，这就如同前面的例子。

还有一种退化的情况，就是让客户端和 Director 融合起来，让客户端直接去操作 Builder，就好像是指导者自己想要给自己构建产品一样。

4. 生成器模式的调用顺序示意图

生成器模式的调用顺序如图 8.3 所示。

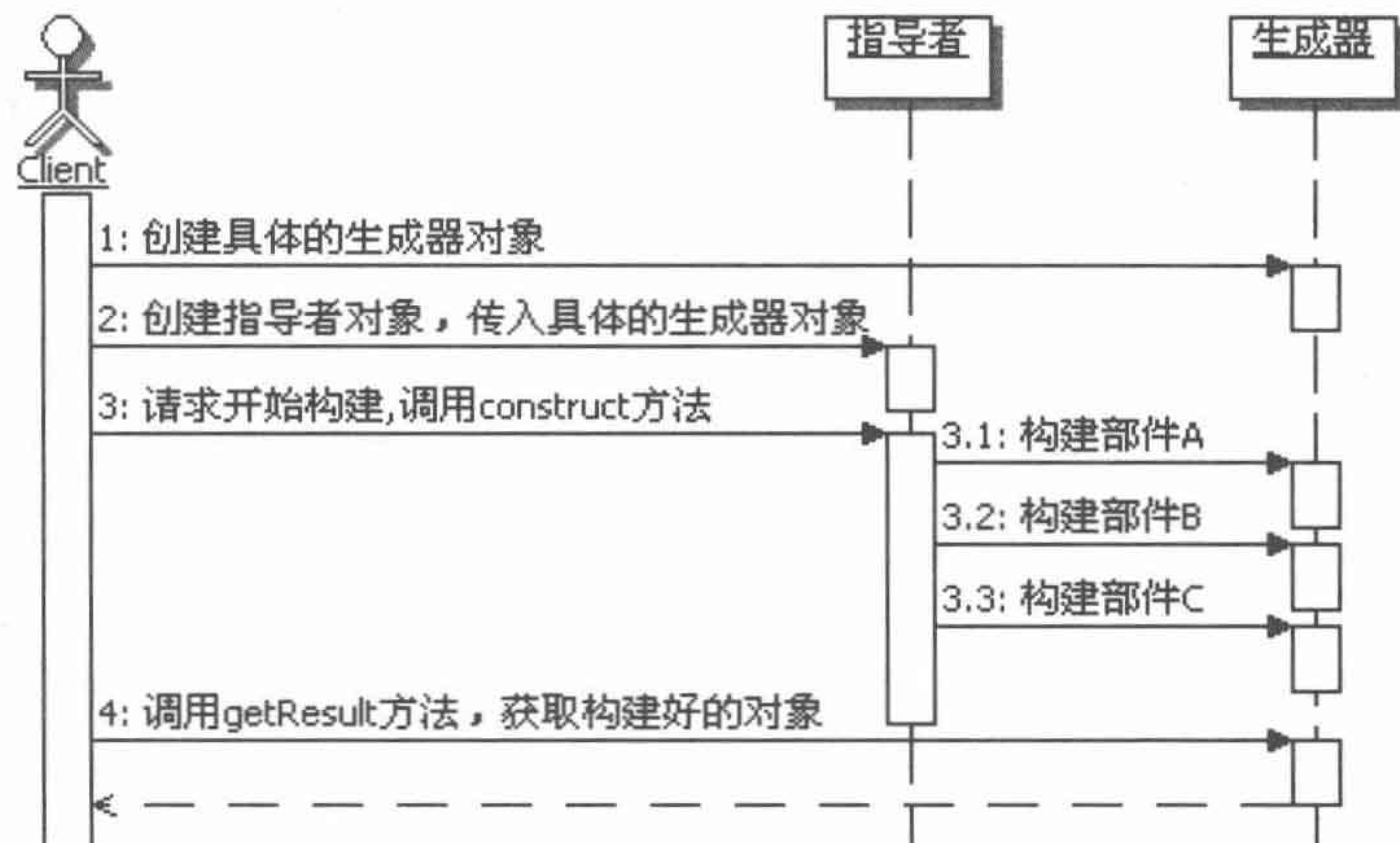


图 8.3　生成器模式的调用顺序示意图

8.3.2　生成器模式的实现

1. 生成器的实现

实际上在 Builder 接口的实现中，每个部件构建的方法里面，除了部件装配外，也可以实现如何具体地创建各个部件对象。也就是说每个方法都可以有两部分功能，一部分是创建部件对象，另一部分是组装部件。

在构建部件的方法里面可以实现选择并创建具体的部件对象，然后再把这个部件对象组装到产品对象中去。这样一来，Builder 就可以和工厂方法配合使用了。

再进一步，如果在实现 Builder 的时候，只有创建对象的功能，而没有组装的功能，那么这个时候的 Builder 实现跟抽象工厂的实现是类似的。

这种情况下，Builder 接口就类似于抽象工厂的接口，Builder 的具体实现就类似于具体的工厂，而且 Builder 接口里面定义的创建各个部件的方法也是有关联的，这些方法是构建一个复杂对象所需要的部件对象。仔细想想，是不是非常类似呢？

2. 指导者的实现

在生成器模式里面，指导者承担的是整体构建算法部分，是相对不变的部分。因此在实现指导者的时候，把变化的部分分离出去是很重要的。

其实指导者分离出去的变化部分，就到了生成器那里，指导者知道整体的构建算法，却不知道如何具体地创建和装配部件对象。

因此真正的指导者实现，并不仅仅是如同前面示例那样，简单地按照一定的顺序调用生成器的方法来生成对象。应该是有较为复杂的算法和运算过程，在运算过程中根据需要，才会调用生成器的方法来生成部件对象。

3. 指导者和生成器的交互

在生成器模式里面，指导者和生成器的交互是通过生成器的 buildPart 方法来完成的。在前面的示例中，指导者和生成器是没有太多相互交互的，指导者仅仅只是简单地调用了一下生成器的方法，在实际开发中，这是远远不够的。

指导者通常会实现比较复杂的算法或者是运算过程，在实际中很可能会有以下的情况：

- 在运行指导者的时候，会按照整体构建算法的步骤进行运算，可能先运行前几步运算，到了某一步骤，需要具体创建某个部件对象了，然后就调用 Builder 中创建相应部件的方法来创建具体的部件。同时，把前面运算得到的数据传递给 Builder，因为在 Builder 内部实现创建和组装部件的时候，可能会需要这些数据。
- Builder 创建完具体的部件对象后，会把创建好的部件对象返回给指导者，指导者继续后续的算法运算，可能会用到已经创建好的对象。
- 如此反复下去，直到整个构建算法运行完成，那么最终的产品对象也就创建好了。

通过上面的描述，可以看出指导者和生成器是需要交互的，方式就是通过生成器方法的参数和返回值，来回地传递数据。事实上，指导者是通过委托的方式来把功能交给生成器去完成。

4. 返回装配好的产品的方法

标准的生成器模式中，在 Builder 实现里面会提供一个返回装配好的产品的方法，在 Builder 接口上是没有的。它考虑的是最终的对象一定要通过部件构建和装配，才算真正创建了，而具体干活的就是 Builder 实现，虽然指导者也参与了，但是指导者是不负责具体的部件创建和组装的，因此客户端是从 Builder 实现里面获取最终装配好的产品。

在 Java 中，我们也可以把这个方法添加到 Builder 接口里面。

5. 关于被构建的产品的接口

在使用生成器模式的时候，大多数情况下是不知道最终构建出来的产品是什么样的，所以在标准的生成器模式里面，一般是不需要对产品定义抽象接口的，因为最终构造的产品千差万别，给这些产品定义公共接口几乎是没有意义的。

8.3.3 使用生成器模式构建复杂对象

考虑这样一个实际应用，要创建一个保险合同的对象，里面很多属性的值都有约束，要求创建出来的对象是满足这些约束规则的。约束规则比如，保险合同通常情况下可以和个人签订，也可以和某个公司签订，但是一份保险合同不能同时与个人和公司签订。这个对象里面有很多类似这样的约束，那么该如何来创建这个对象呢？

要想简洁直观、安全性好，又具有很好的扩展性地来创建这个对象的话，一个较好的选择就是使用 Builder 模式，把复杂的创建过程通过 Builder 来实现。

采用 Builder 模式来构建复杂的对象，通常会对 Builder 模式进行一定的简化，因为目标明确，就是创建某个复杂对象，因此做适当简化会使程序更简洁。大致简化如下。

- 由于是用 Builder 模式来创建某个对象，因此就没有必要再定义一个 Builder 接口，直接提供一个具体的构建器类就可以了。
- 对于创建一个复杂的对象，可能会有很多种不同的选择和步骤，干脆去掉“指导者”，把指导者的功能和 Client 的功能合并起来，也就是说，Client 这个时候就相当于指导者，它来指导构建器类去构建需要的复杂对象。

还是来看看示例会比较清楚。为了实例简单，先不去考虑约束的实现，只是考虑如何通过 Builder 模式来构建复杂对象。

1. 使用 Builder 模式来构建复杂对象，先不考虑带约束

（1）先看一下保险合同的对象。示例代码如下：

```
/**
 * 保险合同的对象
 */
public class InsuranceContract {
    /**
     * 保险合同编号
     */
    private String contractId;
    /**
     * 被保险人员的名称，同一份保险合同，要么跟人员签订，要么跟公司签订
     * 也就是说，"被保险人员"和"被保险公司"这两个属性，不可能同时有值
     */
    private String personName;
    /**
     * 被保险公司的名称
     */
    private String companyName;
    /**
     * 保险开始生效的日期
     */
    private long beginDate;
    /**
     * 保险失效的日期，一定会大于保险开始生效的日期
     */
    private long endDate;
    /**
```

```
     * 示例：其他数据
     */
    private String otherData;

    /**
     * 构造方法，访问级别是同包能访问
     */
    InsuranceContract(ConcreteBuilder builder){
        this.contractId = builder.getContractId();
        this.personName = builder.getPersonName();
        this.companyName = builder.getCompanyName();
        this.beginDate = builder.getBeginDate();
        this.endDate = builder.getEndDate();
        this.otherData = builder.getOtherData();
    }

    /**
     * 示意：保险合同的某些操作
     */
    public void someOperation(){
        System.out.println(
                "Now in Insurance Contract someOperation=="
                +this.contractId);
    }
}
```

> 注意　**注意上例中的构造方法是 default 的访问权限，也就是不希望外部的对象直接通过 new 来构建保险合同对象，另外构造方法传入的是构建器对象，里面包含所有保险合同需要的数据。**

（2）具体的构建器实现的示例代码如下：

```
/**
 * 构造保险合同对象的构建器
 */
public class ConcreteBuilder {
    private String contractId;
    private String personName;
    private String companyName;
```

```
    private long beginDate;
    private long endDate;
    private String otherData;
    /**
     * 构造方法，传入必须要有的参数
     * @param contractId 保险合同编号
     * @param beginDate 保险开始生效的日期
     * @param endDate 保险失效的日期
     */
    public ConcreteBuilder(String contractId,long beginDate
                                              ,long endDate){
        this.contractId = contractId;
        this.beginDate = beginDate;
        this.endDate = endDate;
    }
    /**
     * 选填数据，被保险人员的名称
     * @param personName  被保险人员的名称
     * @return 构建器对象
     */
    public ConcreteBuilder setPersonName(String personName){
        this.personName = personName;
        return this;
    }
    /**
     *  选填数据，被保险公司的名称
     * @param companyName 被保险公司的名称
     * @return 构建器对象
     */
    public ConcreteBuilder setCompanyName(String companyName){
        this.companyName = companyName;
        return this;
    }
    /**
     * 选填数据，其他数据
     * @param otherData 其他数据
     * @return 构建器对象
     */
    public ConcreteBuilder setOtherData(String otherData){
        this.otherData = otherData;
```

```
        return this;
    }
    /**
     * 构建真正的对象并返回
     * @return 构建的保险合同的对象
     */
    public InsuranceContract build(){
        return new InsuranceContract(this);
    }

    public String getContractId() {
        return contractId;
    }
    public String getPersonName() {
        return personName;
    }
    public String getCompanyName() {
        return companyName;
    }
    public long getBeginDate() {
        return beginDate;
    }
    public long getEndDate() {
        return endDate;
    }
    public String getOtherData() {
        return otherData;
    }
}
```


注意

注意上例中，构建器提供了类似于 setter 的方法，来供外部设置需要的参数，为何说是类似于 setter 方法呢？请注意观察，每个这种方法都有返回值，返回的是构建器对象，这样客户端就可以通过连缀的方式来使用 Builder，以创建他们需要的对象。

（3）接下来看看此时的 Client，如何使用上面的构建器来创建保险合同对象。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建构建器
```

```
        ConcreteBuilder builder =
                    new ConcreteBuilder("001",12345L,67890L);
        //设置需要的数据，然后构建保险合同对象
        InsuranceContract contract = builder
            .setPersonName("张三").setOtherData("test").build();
        //操作保险合同对象的方法
        contract.someOperation();
    }
}
```

运行结果如下：

```
Now in Insurance Contract someOperation==001
```

看起来通过 Builder 模式构建对象也很简单，接下来，把约束加上去，看看如何实现？

2. 使用 Builder 模式来构建复杂对象，考虑带约束规则

要带着约束规则构建复杂对象，大致的实现步骤与刚才的实现并没有什么不同，只是需要在刚才的实现上把约束规则添加上去。

通常有两个地方可以添加约束规则。

- 一个是构建器的每一个类似于 setter 的方法，可以在这里进行单个数据的约束规则校验，如果不正确，就抛出 IllegalStateException。
- 另一个是构建器的 build 方法，在创建保险合同对象之前，对所有的数据都可以进行数据的约束规则校验，尤其是那些涉及到几个数据之间的约束关系，在这里校验会比较合适。如果不正确，同样抛出 IllegalStateException。

这里选择在构建器的 build 方法里面进行数据的整体校验。由于其他的代码都没有变化，因此这里就不再赘述了。新的 build 方法的示例代码如下：

```
/**
 * 构建真正的对象并返回
 * @return 构建的保险合同的对象
 */
public InsuranceContract build(){
    if(contractId==null || contractId.trim().length()==0){
        throw new IllegalArgumentException("合同编号不能为空");
    }
    boolean signPerson =
                personName!=null && personName.trim().length()>0;
    boolean signCompany =
            companyName!=null && companyName.trim().length()>0;
```

```
    if(signPerson && signCompany){
        throw new IllegalArgumentException(
                                    "一份保险合同不能同时与人和公司签订");
    }
    if(signPerson==false && signCompany==false){
        throw new IllegalArgumentException(
                                    "一份保险合同不能没有签订对象");
    }
    if(beginDate<=0){
        throw new IllegalArgumentException(
                                    "合同必须有保险开始生效的日期");
    }
    if(endDate<=0){
        throw new IllegalArgumentException(
                                    "合同必须有保险失效的日期");
    }
    if(endDate<=beginDate){
        throw new IllegalArgumentException(
                                    "保险失效的日期必须大于保险生效日期");
    }

    return new InsuranceContract(this);
}
```

可以修改客户端的构建代码，传入不同的数据，看看这些约束规则是否能够正常工作。当然类似的规则还有很多，这里就不去深究了。

3. 进一步，把构建器对象和被构建对象合并

其实，在实际开发中，如果构建器对象和被构建的对象是这样分开的话，可能会导致同包内的对象不使用构建器来构建对象，而是直接去使用 new 来构建对象，这会导致错误；另外，这个构建器的功能就是为了创建被构建的对象，完全可以不用单独一个类。

对于这种情况，重构的手法通常是将类内联化（Inline Class）放到这里来。简单地说就是把构建器对象合并到被构建对象里面去。

还是看看示例会比较清楚。示例代码如下：

```java
public class InsuranceContract {
    private String contractId;
    private String personName;
    private String companyName;
    private long beginDate;
    private long endDate;
    private String otherData;

    /**
     * 构造方法，访问级别是私有的
     */
    private InsuranceContract(ConcreteBuilder builder){
        this.contractId = builder.contractId;
        this.personName = builder.personName;
        this.companyName = builder.companyName;
        this.beginDate = builder.beginDate;
        this.endDate = builder.endDate;
        this.otherData = builder.otherData;
    }

    /**
     * 构造保险合同对象的构建器，作为保险合同的类级内部类
     */
    public static class ConcreteBuilder {
        private String contractId;
        private String personName;
        private String companyName;
        private long beginDate;
        private long endDate;
        private String otherData;
        /**
         * 构造方法，传入必须要有的参数
         * @param contractId 保险合同编号
         * @param beginDate 保险开始生效的日期
         * @param endDate 保险失效的日期
         */
        public ConcreteBuilder(String contractId,
                              long beginDate,long endDate){
            this.contractId = contractId;
            this.beginDate = beginDate;
```

```
        this.endDate = endDate;
    }
    /**
     * 选填数据，被保险人员的名称
     * @param personName  被保险人员的名称
     * @return 构建器对象
     */
    public ConcreteBuilder setPersonName(String personName){
        this.personName = personName;
        return this;
    }
    /**
     *  选填数据，被保险公司的名称
     * @param companyName 被保险公司的名称
     * @return 构建器对象
     */
    public ConcreteBuilder setCompanyName(String companyName){
        this.companyName = companyName;
        return this;
    }
    /**
     * 选填数据，其他数据
     * @param otherData 其他数据
     * @return 构建器对象
     */
    public ConcreteBuilder setOtherData(String otherData){
        this.otherData = otherData;
        return this;
    }
    /**
     * 构建真正的对象并返回
     * @return 构建的保险合同的对象
     */
    public InsuranceContract build(){
        if(contractId==null || contractId.trim().length()==0){
            throw new IllegalArgumentException(
                                        "合同编号不能为空");
        }
        boolean signPerson =
            personName!=null && personName.trim().length()>0;
```

```
            boolean signCompany =
              companyName!=null && companyName.trim().length()>0;
            if(signPerson && signCompany){
                throw new IllegalArgumentException(
                            "一份保险合同不能同时与人和公司签订");
            }
            if(signPerson==false && signCompany==false){
                throw new IllegalArgumentException(
                            "一份保险合同不能没有签订对象");
            }
            if(beginDate<=0){
                throw new IllegalArgumentException(
                            "合同必须有保险开始生效的日期");
            }
            if(endDate<=0){
                throw new IllegalArgumentException(
                            "合同必须有保险失效的日期");
            }
            if(endDate<=beginDate){
                throw new IllegalArgumentException(
                            "保险失效的日期必须大于保险生效日期");
            }

            return new InsuranceContract(this);
        }
    }

    /**
     * 示意：保险合同的某些操作
     */
    public void someOperation(){
        System.out.println(
                    "Now in Insurance Contract someOperation=="
                    +this.contractId);
    }
}
```

通过上面的示例可以看出，这种实现方式会更简单和直观。

此时客户端的写法也发生了一点变化，主要就是创建构造器的地方需要变化，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建构建器
        InsuranceContract.ConcreteBuilder builder =
      new InsuranceContract.ConcreteBuilder("001",12345L,67890L);
        //设置需要的数据，然后构建保险合同对象
        InsuranceContract contract = builder
                .setPersonName("张三").setOtherData("test").build();
        //操作保险合同对象的方法
        contract.someOperation();
    }
}
```

测试一下看看，体会一下使用 Builder 模式来构建复杂对象，尤其是在构造方法需要大量参数，或者是构建带约束规则的复杂对象时的使用方式。

8.3.4 生成器模式的优点

- 松散耦合

  生成器模式可以用同一个构建算法构建出表现上完全不同的产品，实现产品构建和产品表现上的分离。生成器模式正是把产品构建的过程独立出来，使它和具体产品的表现松散耦合，从而使得构建算法可以复用，而具体产品表现也可以灵活地、方便地扩展和切换。

- 可以很容易地改变产品的内部表示

  在生成器模式中，由于 Builder 对象只是提供接口给 Director 使用，那么具体的部件创建和装配方式是被 Builder 接口隐藏了的，Director 并不知道这些具体的实现细节。这样一来，要想改变产品的内部表示，只需要切换 Builder 的具体实现即可，不用管 Director，因此变得很容易。

- 更好的复用性

  生成器模式很好地实现了构建算法和具体产品实现的分离。这样一来，使得构建产品的算法可以复用。同样的道理，具体产品的实现也可以复用，同一个产品的实现，可以配合不同的构建算法使用。

8.3.5 思考生成器模式

1. 生成器模式的本质

> **生成器模式的本质**：分离整体构建算法和部件构造。

构建一个复杂的对象，本来就有构建的过程，以及构建过程中具体的实现。生成器模式就是用来分离这两个部分，从而使得程序结构更松散、扩展更容易、复用性更好，同时也会使得代码更清晰，意图更明确。

虽然在生成器模式的整体构建算法中，会一步一步引导 Builder 来构建对象，但这并不是说生成器主要就是用来实现分步骤构建对象的。**生成器模式的重心还是在于分离整体构建算法和部件构造，而分步骤构建对象不过是整体构建算法的一个简单表现，或者说是一个附带产物。**

2. 何时选用生成器模式

建议在以下情况中选用生成器模式。

- 如果创建对象的算法，应该独立于该对象的组成部分以及它们的装配方式时。
- 如果同一个构建过程有着不同的表示时。

8.3.6　相关模式

- 生成器模式和工厂方法模式

  这两个模式可以组合使用。

  生成器模式的 Builder 实现中，通常需要选择具体的部件实现。一个可行的方案就是实现成为工厂方法，通过工厂方法来获取具体的部件对象，然后再进行部件的装配。

- 生成器模式和抽象工厂模式

  这两个模式既相似又有区别，也可以组合使用。

  先说相似性，这个在 8.3.2 小节的第一个小题目里面已经详细讲述了，这里就不再重复了。

  再说说区别：抽象工厂模式的主要目的是创建产品簇，这个产品簇里面的单个产品就相当于是构成一个复杂对象的部件对象，抽象工厂对象创建完成后就立即返回整个产品簇；而生成器模式的主要目的是按照构造算法，一步一步来构建一个复杂的产品对象，通常要等到整个构建过程结束以后，才会得到最终的产品对象。

  事实上，这两个模式是可以组合使用的。在生成器模式的 Builder 实现中，需要创建各个部件对象，而这些部件对象是有关联的，通常是构成一个复杂对象的部件对象。也就是说，Builder 实现中，需要获取构成一个复杂对象的产品簇，那自然就可以使用抽象工厂模式来实现。这样一来，由抽象工厂模式负责了部件对象的创建，Builder 实现里面则主要负责产品对象整体的构建了。

- 生成器模式和模板方法模式

  这也是两个非常类似的模式。初看之下，不会觉得这两个模式有什么关联。但是仔细一思考，却发现两个模式在功能上很类似。模板方法模式主要是用来定义算法的骨架，把算法中某些步骤延迟到子类中实现。再想想生成器模式，Director 用来定义整体的构建算法，把算法中某些涉及到具体部件对象的创建

和装配的功能，委托给具体的 Builder 来实现。

虽然生成器不是延迟到子类，是委托给 Builder，但那只是具体实现方式上的差别，从实质上看两个模式很类似，都是定义一个固定的算法骨架，然后把算法中的某些具体步骤交给其他类来完成，都能实现整体算法步骤和某些具体步骤实现的分离。

当然这两个模式也有很大的区别，首先是模式的目的，生成器模式是用来构建复杂对象的，而模板方法是用来定义算法骨架，尤其是一些复杂的业务功能的处理算法的骨架；其次是模式的实现，生成器模式是采用委托的方法，而模板方法采用的是继承的方式；另外从使用的复杂度上，生成器模式需要组合 Director 和 Builder 对象，然后才能开始构建，要等构建完后才能获得最终的对象，而模板方法就没有这么麻烦，直接使用子类对象即可。

- 生成器模式和组合模式

  这两个模式可以组合使用。

  对于复杂的组合结构，可以使用生成器模式来一步一步构建。

第 9 章　原型模式（Prototype）

9.1 场景问题

9.1.1 订单处理系统

考虑这样一个实际应用：订单处理系统。

现在有一个订单处理的系统，里面有一个保存订单的业务功能。在这个业务功能中，客户有这样一个需求：每当订单的预定产品数量超过1000的时候，就需要把订单拆成两份订单来保存。如果拆成两份订单后，还是超过1000，那就继续拆分，直到每份订单的预定产品数量不超过1000。至于为什么要拆分，原因是方便进行订单的后续处理，后续是由人工来处理，每个人工工作小组的处理能力上限是1000。

根据业务，目前的订单类型被分成两种：一种是个人订单，一种是公司订单。现在想要实现一个通用的订单处理系统，也就是说，不管具体是什么类型的订单，都要能够正常地处理。

该怎么实现呢？

9.1.2 不用模式的解决方案

来分析上面要求实现的功能，有朋友会想，这很简单嘛，一共就一个功能，没什么困难的，真的是这样吗？下面来尝试着实现看看。

（1）定义订单接口。

首先，要想实现通用的订单处理，而不关心具体的订单类型，那么很明显，订单处理的对象应该面向一个订单的接口或是一个通用的订单对象来编程，这里就选用面向订单的接口来处理。先把这个订单接口定义出来。示例代码如下：

```java
/**
 * 订单的接口
 */
public interface OrderApi {
 /**
  * 获取订单产品数量
  * @return 订单中产品数量
  */
 public int getOrderProductNum();
 /**
  * 设置订单产品数量
  * @param num 订单产品数量
  */
 public void setOrderProductNum(int num);
}
```

（2）既然定义好了订单的接口，那么接下来把各种类型的订单实现出来。

先来看个人的订单实现。示例代码如下：

```
/**
 * 个人订单对象
 */
public class PersonalOrder implements OrderApi{
    /**
     * 订购人员姓名
     */
    private String customerName;
    /**
     * 产品编号
     */
    private String productId;
    /**
     * 订单产品数量
     */
    private int orderProductNum = 0;

    public int getOrderProductNum() {
        return this.orderProductNum;
    }
    public void setOrderProductNum(int num) {
        this.orderProductNum = num;
    }
    public String getCustomerName() {
        return customerName;
    }
    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
    public String toString(){
        return "本个人订单的订购人是="+this.customerName
                +"，订购产品是="+this.productId+"，订购数量为="
```

```
                    +this.orderProductNum;
}
}
```

再看看企业订单的实现。示例代码如下：

```
/**
 * 企业订单对象
 */
public class EnterpriseOrder implements OrderApi{
    /**
     * 企业名称
     */
    private String enterpriseName;
    /**
     * 产品编号
     */
    private String productId;
    /**
    * 订单产品数量
    */
    private int orderProductNum = 0;

    public int getOrderProductNum() {
        return this.orderProductNum;
    }
    public void setOrderProductNum(int num) {
        this.orderProductNum = num;
    }
    public String getEnterpriseName() {
        return enterpriseName;
    }
    public void setEnterpriseName(String enterpriseName) {
        this.enterpriseName = enterpriseName;
    }
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
    public String toString(){
```

```
        return "本企业订单的订购企业是="+this.enterpriseName
                +"，订购产品是="+this.productId+"，订购数量为="
                                +this.orderProductNum;
    }
}
```

有些朋友看到这里，可能会有这样的疑问：看上去上面两种类型的订单对象，仅仅是一个数据封装的对象，而且还有一些数据是相同的，为何不抽出一个父类来，把共同的数据定义在父类里面呢？

> **延伸** **这里有两个考虑，一个是：这里仅仅是一个示意，实际情况远比这复杂，实际开发中不会仅仅是数据封装对象这么简单。另外一个是：为了后续示例的重点突出，这里要学习的是原型模式，因此就没有抽取父类，以免对象层级过多，影响主题的展示。**

（3）实现好了订单对象，接下来看看如何实现通用的订单处理。先把订单处理的对象大概定义出来。示例代码如下：

```
/**
 * 处理订单的业务对象
 */
public class OrderBusiness {
    /**
     * 创建订单的方法
     * @param order 订单的接口对象
     */
    public void saveOrder(OrderApi order){
//等待具体实现
    }
}
```

现在的中心任务就是要来实现这个 saveOrder 的方法，传入的参数是一个订单的接口对象，这个方法要实现的功能是：根据业务要求，当订单的预定产品数量超过 1000 的时候，就需要把订单拆成两份订单。

那好，来尝试着实现一下。因为预定的数量可能会很大，因此采用一个 while 循环来处理，直到拆分后订单的数量不超过 1000。先把实现的思路写出来。示例代码如下：

```
public class OrderBusiness {
    public void saveOrder(OrderApi order){
    //1：判断当前的预定产品数量是否大于 1000
    while(order.getOrderProductNum() > 1000){
        //2：如果大于，还需要继续拆分
        //2.1 再新建一份订单，跟传入的订单除了数量不一样外，其他都相同
```

```
            OrderApi newOrder = null;
```

问题产生了：如何新建一份订单

```
        }
    }
}
```

大家会发现，刚写到第二步就写不下去了，为什么呢？因为现在判断需要拆分订单，也就是需要新建一个订单对象，可是订单处理对象面对的是订单的接口，它根本就不知道现在订单具体的类型，也不知道具体的订单实现，所以无法创建出新的订单对象来，也就无法实现订单拆分的功能了。

（4）一个简单的解决办法。

有朋友提供了这么一个解决的思路，他说：不就是在 saveOrder 方法里面不知道具体的类型，从而导致无法创建对象吗？很简单，使用 instanceof 来判断不就可以了，他还给出了他的实现示意。示意代码如下：

```
public class OrderBusiness {
public void saveOrder(OrderApi order){
    while(order.getOrderProductNum() > 1000)
        //定义一个表示被拆分出来的新订单对象
        OrderApi newOrder = null;
        if(order instanceof PersonalOrder){
            //创建相应的订单对象
            PersonalOrder p2 = new PersonalOrder();
            //然后进行赋值等，省略了
            //再设置给newOrder
            newOrder = p2;
        }else if(order instanceof EnterpriseOrder){
            //创建相应的订单对象
            EnterpriseOrder e2 = new EnterpriseOrder();
            //然后进行赋值等，省略了
            //再设置给newOrder
            newOrder = e2;
        }
        //进行拆分和其他业务功能处理，省略了
    }
}
}
```

好像能解决问题，对吧。那我们就来按照他提供的思路，把这个通用的订单处理对象实现出来。示例代码如下：

```
/**
```

```
 * 处理订单的业务对象
 */
public class OrderBusiness {
    /**
     * 创建订单的方法
     * @param order 订单的接口对象
     */
    public void saveOrder(OrderApi order){
        //根据业务要求，当订单预定产品数量超过 1000 时，就要把订单拆成两份订单
        //当然如果要做好，这里的 1000 应该做成常量，这么做是为了演示简单

        //1：判断当前的预定产品数量是否大于 1000
        while(order.getOrderProductNum() > 1000){
            //2：如果大于，还需要继续拆分
            //2.1 再新建一份订单，跟传入的订单除了数量不一样外，其他都相同
            OrderApi newOrder = null;
            if(order instanceof PersonalOrder){
                //创建相应的新的订单对象
                PersonalOrder p2 = new PersonalOrder();
                //然后进行赋值，但是产品数量为 1000
                PersonalOrder p1 = (PersonalOrder)order;
                p2.setCustomerName(p1.getCustomerName());
                p2.setProductId(p1.getProductId());
                p2.setOrderProductNum(1000);
                //再设置给 newOrder
                newOrder = p2;
            }else if(order instanceof EnterpriseOrder){
                //创建相应的订单对象
                EnterpriseOrder e2 = new EnterpriseOrder();
                //然后进行赋值，但是产品数量为 1000
                EnterpriseOrder e1 = (EnterpriseOrder)order;
                e2.setEnterpriseName(e1.getEnterpriseName());
                e2.setProductId(e1.getProductId());
                e2.setOrderProductNum(1000);
                //再设置给 newOrder
                newOrder = e2;
            }

            //2.2 原来的订单保留，把数量设置成减少 1000
            order.setOrderProductNum(
```

```
                                order.getOrderProductNum()-1000);

                //然后是业务功能处理，省略了，打印输出，看一下
                System.out.println("拆分生成订单=="+newOrder);
            }
            //3：不超过1000，那就直接业务功能处理，省略了，打印输出，看一下
            System.out.println("订单=="+order);
        }
}
```

（5）编写客户端程序来测试一下。示例代码如下：

```
public class OrderClient {
    public static void main(String[] args) {
        //创建订单对象，这里为了演示简单，直接new了
        PersonalOrder op = new PersonalOrder();
        //设置订单数据
        op.setOrderProductNum(2925);
        op.setCustomerName("张三");
        op.setProductId("P0001");

        //这里获取业务处理的类，也直接new了，为了简单，连业务接口都没有做
        OrderBusiness ob = new OrderBusiness();
        //调用业务来保存订单对象
        ob.saveOrder(op);
    }
}
```

运行结果如下：

```
拆分生成订单==本个人订单的订购人是=张三，订购产品是=P0001，订购数量为=1000
拆分生成订单==本个人订单的订购人是=张三，订购产品是=P0001，订购数量为=1000
订单==本个人订单的订购人是=张三，订购产品是=P0001，订购数量为=925
```

根据订单中订购产品的数量，一份订单被拆分成了三份。

同样的，你还可以传入企业订单，看看是否能正常满足功能要求。

9.1.3 有何问题

看起来，上面的实现确实不难，好像也能够通用地进行订单处理，而不需要关心订单的类型和具体实现这样的功能。

仔细想想，真的没有关心订单的类型和具体实现吗？答案是“否定的”。

事实上，在实现订单处理的时候，上面的实现是按照订单的类型和具体实现来处理的，就是 instanceof 的那一段。有朋友可能会问，这样实现有何不可吗？

这样的实现有以下几个问题。

- 既然想要实现通用的订单处理，那么对于订单处理的实现对象，是不应该知道订单的具体实现的，更不应该依赖订单的具体实现。但是上面的实现中，很明显订单处理的对象依赖了订单的具体实现对象。
- 这种实现方式另外一个问题就是：难以扩展新的订单类型。假如现在要加入一个大客户专用订单的类型，那么就需要修改订单处理的对象，要在里面添加对新的订单类型的支持，这不能算做通用处理。

因此，上面的实现是不太好的，把上面的问题再抽象描述一下：**已经有了某个对象实例后，如何能够快速简单地创建出更多的这种对象？**

比如上面的问题，就是已经有了订单接口类型的对象实例，然后在方法中需要创建出更多的这种对象。怎么解决呢？

9.2　解 决 方 案

9.2.1　使用原型模式来解决问题

用来解决上述问题的一个合理的解决方案就是原型模式（Prototype）。那么什么是原型模式呢？

1. 原型模式的定义

> 用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

2. 应用原型模式来解决问题的思路

仔细分析上面的问题，在 saveOrder 方法里面，已经有了订单接口类型的对象实例，是从外部传入的，但是这里只是知道这个实例对象的种类是订单的接口类型，并不知道其具体的实现类型，也就是不知道它到底是个人订单还是企业订单，但是现在需要在这个方法里面创建一个这样的订单对象，看起来就像是要通过接口来创建对象一样。

原型模式就可以解决这样的问题。原型模式会要求对象实现一个可以“克隆”自身的接口，这样就可以通过拷贝或者是克隆一个实例对象本身来创建一个新的实例。如果把这个方法定义在接口上，看起来就像是通过接口来创建了新的接口对象。

这样一来，通过原型实例创建新的对象，就不再需要关心这个实例本身的类型，也不关心它的具体实现，只要它实现了克隆自身的方法，就可以通过这个方法来获取新的对象，而无须再去通过 new 来创建。

9.2.2 原型模式的结构和说明

原型模式的结构如图 9.1 所示。

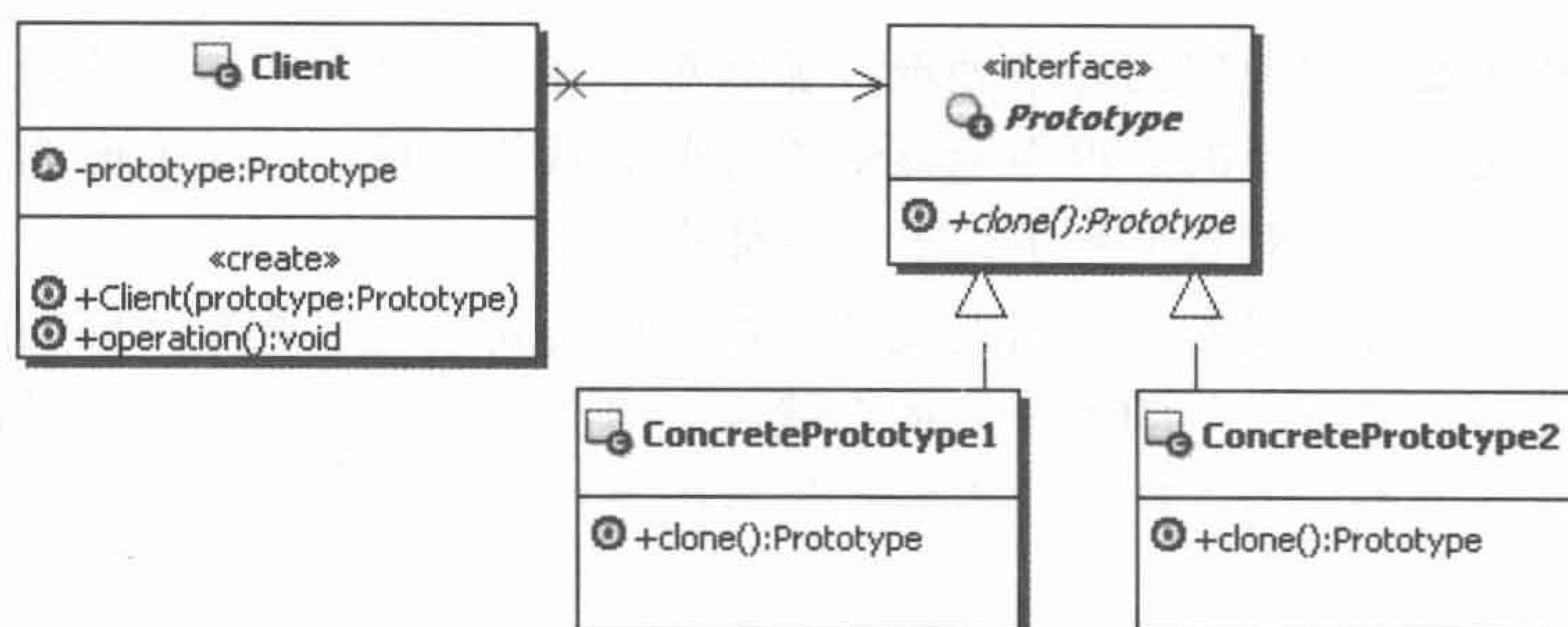


图 9.1　原型模式结构示意图

- Prototype：声明一个克隆自身的接口，用来约束想要克隆自己的类，要求它们都要实现这里定义的克隆方法。
- ConcretePrototype：实现 Prototype 接口的类，这些类真正实现了克隆自身的功能。
- Client：使用原型的客户端，首先要获取到原型实例对象，然后通过原型实例克隆自身来创建新的对象实例。

9.2.3 原型模式示例代码

（1）先来看看原型接口的定义。示例代码如下。

```
/**
 * 声明一个克隆自身的接口
 */
public interface Prototype {
    /**
     * 克隆自身的方法
     * @return 一个从自身克隆出来的对象
     */
    public Prototype clone();
}
```

（2）接下来看看具体的原型实现对象。示例代码如下：

```
/**
 * 克隆的具体实现对象
 */
public class ConcretePrototype1 implements Prototype {
    public Prototype clone() {
        //最简单的克隆，新建一个自身对象，由于没有属性，就不再复制值了
```

```
        Prototype prototype = new ConcretePrototype1();
        return prototype;
    }
}
/**
 * 克隆的具体实现对象
 */
public class ConcretePrototype2 implements Prototype {
    public Prototype clone() {
        //最简单的克隆，新建一个自身对象，由于没有属性，就不再复制值了
        Prototype prototype = new ConcretePrototype2();
        return prototype;
    }
}
```

为了跟上面原型模式的结构示意图保持一致，因此这两个具体的原型实现对象。都没有定义属性。事实上，在实际使用原型模式的应用中，原型对象多是有属性的，克隆原型的时候也是需要克隆原型对象的属性的，特此说明一下。

（3）再看看使用原型的客户端。示例代码如下：

```
/**
 * 使用原型的客户端
 */
public class Client {
    /**
     * 持有需要使用的原型接口对象
     */
    private Prototype prototype;
    /**
     * 构造方法，传入需要使用的原型接口对象
     * @param prototype 需要使用的原型接口对象
     */
    public Client(Prototype prototype){
        this.prototype = prototype;
    }
    /**
     * 示意方法，执行某个功能操作
     */
    public void operation(){
        //需要创建原型接口的对象
        Prototype newPrototype = prototype.clone();
```

```
    }
}
```

9.2.4 使用原型模式重写示例

要使用原型模式来重写示例，先要在订单的接口上定义出克隆的接口，然后要求各个具体的订单对象克隆自身，这样就可以解决：在订单处理对象里面通过订单接口来创建新的订单对象的问题。

使用原型模式来重写示例的结构如图 9.2 所示：

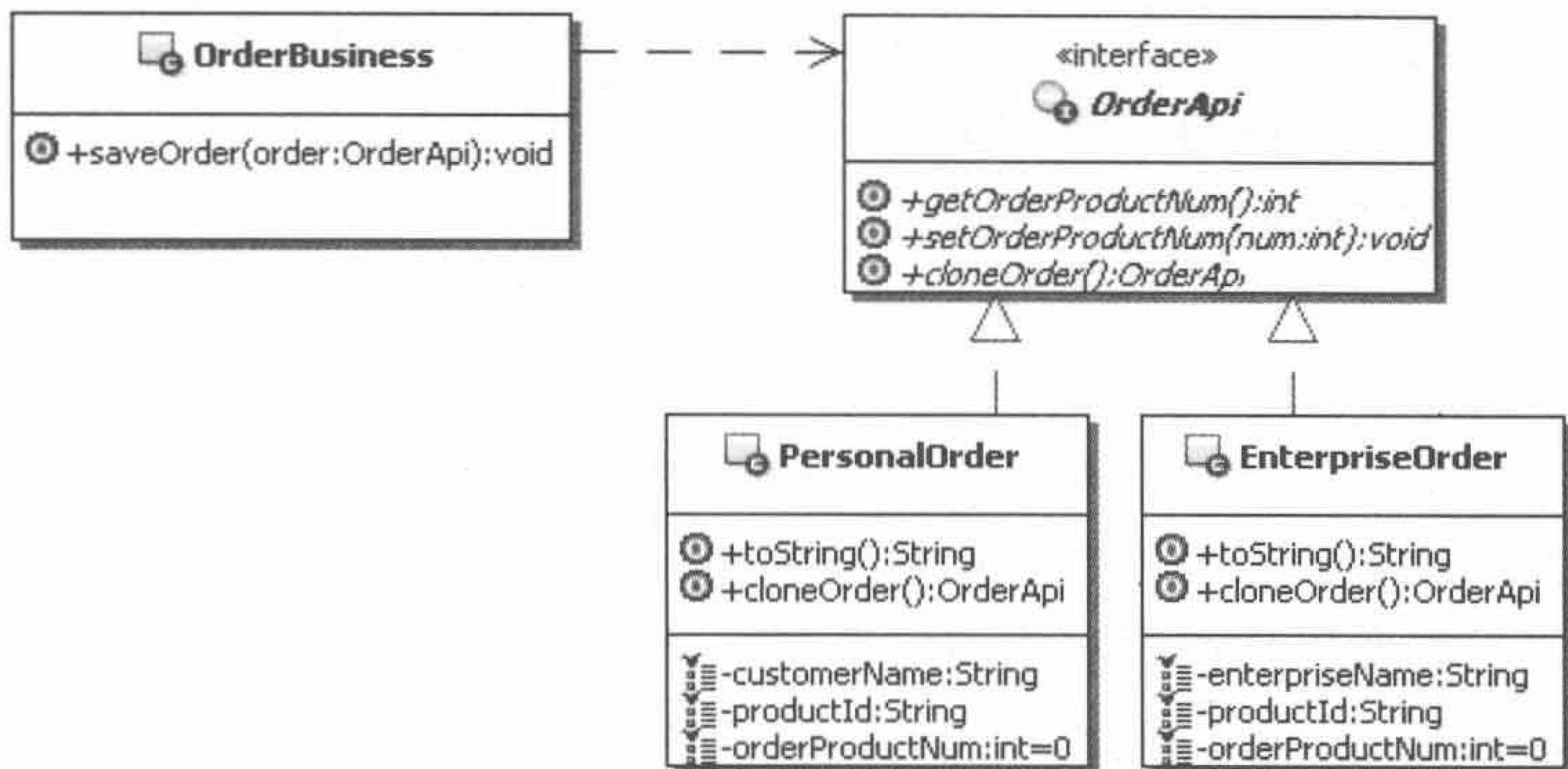


图 9.2 使用原型模式来重写示例的结构示意图

下面一起来看看具体的实现。

1. 复制谁和谁来复制的问题

有了一个对象实例，要快速地创建和它一样的实例，最简单的办法就是复制？这里又有两个小的问题：

- 复制谁呢？当然是复制这个对象实例，复制实例的意思是连带着数据一起复制。
- 谁来复制呢？应该让这个类的实例自己来复制，自己复制自己。

可是每个对象不会那么听话，自己去实现复制自己的。于是原型模式决定对这些对象实行强制要求，给这些对象定义一个接口，在接口里面定义一个方法，这个方法用来要求每个对象实现自己复制自己。

由于现在存在订单的接口，因此就把这个要求克隆自身的方法定义在订单的接口里面。示例代码如下：

```
/**
 * 订单的接口，声明了可以克隆自身的方法
 */
public interface OrderApi {
    public int getOrderProductNum();
    public void setOrderProductNum(int num);
    /**
     * 克隆方法
```

```
     * @return 订单原型的实例
     */
    public OrderApi cloneOrder();
}
```

2. 如何克隆

定义好了克隆的接口，那么在订单的实现类里面，就得让它实现这个接口，并具体地实现这个克隆方法。新的问题出来了，如何实现克隆呢？

很简单，只要先 new 一个自己对象的实例，然后把自己实例中的数据取出来，设置到新的对象实例中去，就可以完成实例的复制，复制的结果就是有了一个同自身一模一样的实例。

有的朋友可能会说：不用那么费劲吧，直接返回本实例不就可以了？例如：

```
public Object clone(){
    return this;
}
```

> 注意。 **请注意，这是一种典型的错误，这么做，每次克隆，客户端获取的其实都是同一个实例，都是指向同一个内存空间的，对克隆出来的对象实例的修改会影响到原型对象实例。**

那么应该怎么克隆呢？最基本的做法就是新建一个类实例，然后把所有属性的值复制到新的实例中。

先看看个人订单对象的实现。示例代码如下：

```
/**
 * 个人订单对象
 */
public class PersonalOrder implements OrderApi{
    private String customerName;
    private String productId;
    private int orderProductNum = 0;

    public int getOrderProductNum() {
        return this.orderProductNum;
    }
    public void setOrderProductNum(int num) {
        this.orderProductNum = num;
    }
    public String getCustomerName() {
        return customerName;
    }
```

```
    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
    public String toString(){
        return "本个人订单的订购人是="+this.customerName
                +"，订购产品是="+this.productId+"，订购数量为="
                                    +this.orderProductNum;
    }
    public OrderApi cloneOrder() {
        //创建一个新的订单，然后把本实例的数据复制过去
        PersonalOrder order = new PersonalOrder();
        order.setCustomerName(this.customerName);
        order.setProductId(this.productId);
        order.setOrderProductNum(this.orderProductNum);

        return order;
    }
}
```

接下来看看企业订单的具体实现。示例代码如下：

```
/**
 * 企业订单对象
 */
public class EnterpriseOrder implements OrderApi{
    private String enterpriseName;
    private String productId;
    private int orderProductNum = 0;
    public int getOrderProductNum() {
        return this.orderProductNum;
    }
    public void setOrderProductNum(int num) {
        this.orderProductNum = num;
    }
    public String getEnterpriseName() {
        return enterpriseName;
```

```
    }
    public void setEnterpriseName(String enterpriseName) {
        this.enterpriseName = enterpriseName;
    }
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
    public String toString(){
        return "本企业订单的订购企业是="+this.enterpriseName
               +"，订购产品是="+this.productId+"，订购数量为="
                                  +this.orderProductNum;
    }
    public OrderApi cloneOrder() {
        //创建一个新的订单，然后把本实例的数据复制过去
        EnterpriseOrder order = new EnterpriseOrder();
        order.setEnterpriseName(this.enterpriseName);
        order.setProductId(this.productId);
        order.setOrderProductNum(this.orderProductNum);
        return order;
    }
}
```

3. 使用克隆方法

这里使用订单接口的克隆方法的是订单的处理对象，也就是说，订单的处理对象就相当于原型模式结构中的 Client。

当然，客户端在调用 clone 方法之前，还需要先获得相应的实例对象，有了实例对象，才能调用该实例对象的 clone 方法。

> 注意　**这里使用克隆方法的时候，和标准的原型实现有一些不同，在标准的原型实现的示例代码里面，客户端是持有需要克隆的对象，而这里变化成了通过方法传入需要使用克隆的对象，这点大家注意一下。**

示例代码如下：

```
public class OrderBusiness {
    /**
     * 创建订单的方法
     * @param order 订单的接口对象
     */
```

```
    public void saveOrder(OrderApi order){
        //1：判断当前的预定产品数量是否大于 1000
        while(order.getOrderProductNum() > 1000){
        //2：如果大于，还需要继续拆分
        //2.1 再新建一份订单，跟传入的订单除了数量不一样外，其他都相同
        OrderApi newOrder = order.cloneOrder();
        //然后进行赋值，产品数量为 1000
        newOrder.setOrderProductNum(1000);

        //2.2 原来的订单保留，把数量设置成减少 1000
        order.setOrderProductNum(
                order.getOrderProductNum()-1000);

        //然后是业务功能处理，省略了，打印输出，看一下
        System.out.println("拆分生成订单=="+newOrder);
        }
        //3：不超过，那就直接业务功能处理，省略了，打印输出，看一下
        System.out.println("订单=="+order);
    }
}
```

客户端的测试代码和前面的示例是完全一样的，这里就不再赘述。运行一下，看看运行的效果，享受一下克隆的乐趣。

在上面的例子中，在订单处理对象的保存订单方法里面的这句话"OrderApi newOrder = order.cloneOrder();"，就用一个订单的原型实例来指定了对象的种类，然后通过克隆这个原型实例来创建出了一个新的对象实例。

看到这里，可能有些朋友会认为：Java 的 Object 里面本身就有 clone 方法，还用搞得这么麻烦吗？

虽然 Java 里面有 clone 方法，上面这么做还是很有意义的，可以更好地、更完整地体会原型设计模式。当然，后面会讲述如何使用 Java 里面的 clone 方法来实现克隆。

9.3 模式讲解

9.3.1 认识原型模式

1. 原型模式的功能

原型模式的功能实际上包含两个方面：

- 一个是通过克隆来创建新的对象实例；
- 另一个是为克隆出来的新的对象实例复制原型实例属性的值。

原型模式要实现的主要功能就是：通过克隆来创建新的对象实例。一般来讲，新创建出来的实例的数据是和原型实例一样的。但是具体如何实现克隆，需要由程序自行实现，原型模式并没有统一的要求和实现算法。

2. 原型与 new

原型模式从某种意义上说，就像是 new 操作，在前面的例子实现中，克隆方法就是使用 new 来实现的。但请注意，只是“类似于 new”而不是“就是 new”。

克隆方法和 new 操作最明显的不同就在于：new 一个对象实例，一般属性是没有值的，或者是只有默认值；如果是克隆得到的一个实例，通常属性是有值的，属性的值就是原型对象实例在克隆的时候，原型对象实例的属性的值。

3. 原型实例和克隆的实例

原型实例和克隆出来的实例，本质上是不同的实例，克隆完成后，它们之间是没有关联的，如果克隆完成后，克隆出来的实例的属性值发生了改变，是不会影响到原型实例的。下面写个示例来测试一下。示例代码如下：

```
public class Client {
public static void main(String[] args) {
    //先创建原型实例
    OrderApi oa1 = new PersonalOrder();

    //设置原型实例的订单数量的值
    oa1.setOrderProductNum(100);
    //为了简单，这里仅仅输出数量
   System.out.println("这是第一次获取的对象实例==="
                          +oa1.getOrderProductNum());

    //通过克隆来获取新的实例
   OrderApi oa2 = (OrderApi)oa1.cloneOrder();
    //修改它的数量
    oa2.setOrderProductNum(80);
    //输出克隆出来的对象的值
    System.out.println("输出克隆出来的实例==="
                          +oa2.getOrderProductNum());

    //再次输出原型实例的值
    System.out.println("再次输出原型实例==="
                          +oa1.getOrderProductNum());
 }
}
```

运行一下，看看结果：

```
这是第一次获取的对象实例===100
输出克隆出来的实例===80
再次输出原型实例===100
```

仔细观察上面的结果，会发现原型实例和克隆出来的实例是完全独立的，也就是它们指向不同的内存空间。因为克隆出来的实例的值已经被改变了，而原型实例的值还是原来的值，并没有变化，这就说明两个实例对应的是不同的内存空间。

4. 原型模式的调用顺序示意图

原型模式的调用顺序如图 9.3 所示。

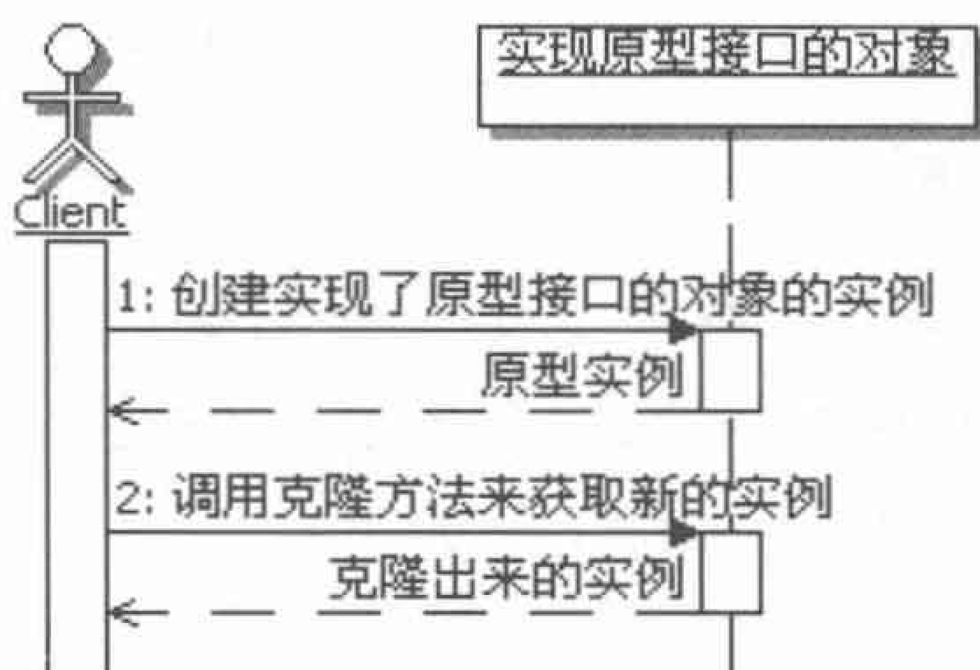


图 9.3　原型模式的调用顺序示意图

9.3.2　Java 中的克隆方法

在 Java 语言中已经提供了 clone 方法，定义在 Object 类中。关于 Java 中 clone 方法的知识，这里不再赘述，下面看看怎么使用 Java 里面的克隆方法来实现原型模式。

需要克隆功能的类，只需要实现 java.lang.Cloneable 接口，这个接口没有需要实现的方法，是一个标识接口。因此在前面的实现中，把订单接口中的克隆方法去掉，现在直接实现 Java 中的接口就可以了。新的订单接口实现，示例代码如下：

```
public interface OrderApi {
    public int getOrderProductNum();
    public void setOrderProductNum(int num);
    public OrderApi cloneOrder();
}
```

另外在具体的订单实现对象里面，实现方式上会有一些改变，个人订单和企业订单的克隆实现是类似的，因此示范一个就可以了。来看看个人订单的实现吧，示例代码如下：

```
/**
 * 个人订单对象，利用Java的Clone功能
 */
public class PersonalOrder implements Cloneable , OrderApi{
    private String customerName;
    private String productId;
    private int orderProductNum = 0;
    public int getOrderProductNum() {
```

```
        return this.orderProductNum;
    }
    public void setOrderProductNum(int num) {
        this.orderProductNum = num;
    }
    public String getCustomerName() {
        return customerName;
    }
    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
    public String toString(){
        return "本个人订单的订购人是="+this.customerName
                +"，订购产品是="+this.productId+"，订购数量为="
                +this.orderProductNum;
    }
    public OrderApi cloneOrder() {
        //创建一个新的订单，然后把本实例的数据复制过去
        PersonalOrder order = new PersonalOrder();
        order.setCustomerName(this.customerName);
        order.setProductId(this.productId);
        order.setOrderProductNum(this.orderProductNum);
        return order;
    }
    public Object clone(){
        //克隆方法的真正实现，直接调用父类的克隆方法就可以了
        Object obj = null;
        try {
            obj = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return obj;
    }
```

原有的克隆实现删除掉

注意这里实现的变化

```
}
```

看起来，比完全由自己实现原型模式要稍稍简单点，是否好用呢？还是测试一下，看看效果。客户端与上一个示例相比，作了两点修改。

- 一个是原来的“OrderApi oa1 = new PersonalOrder();”这句话，要修改成“PersonalOrder oa1 = new PersonalOrder();”。原因是现在的接口上并没有克隆的方法，因此需要修改成原型的类型。
- 另外一个是“通过克隆来获取新的实例”的实现，需要修改成使用原型来调用在 Object 里面定义的 clone()方法了，不再是调用原来的 cloneOrder()了。

看看测试用的代码。示例代码如下：

```
public class Client {
public static void main(String[] args) {
    //先创建原型实例
    PersonalOrder oa1 = new PersonalOrder();
    //设置原型实例的订单数量的值
    oa1.setOrderProductNum(100);
    System.out.println("这是第一次获取的对象实例==="
                                    +oa1.getOrderProductNum());
    //通过克隆来获取新的实例
    PersonalOrder oa2 = (PersonalOrder)oa1.clone();
    oa2.setOrderProductNum(80);
    System.out.println("输出克隆出来的实例==="
                                    +oa2.getOrderProductNum());
    //再次输出原型实例的值
    System.out.println("再次输出原型实例==="
                                    +oa1.getOrderProductNum());
}
}
```

运行一下，测试看看。

9.3.3 浅度克隆和深度克隆

无论你是自己实现克隆方法，还是采用 Java 提供的克隆方法，都存在一个浅度克隆和深度克隆的问题，那么什么是浅度克隆？什么是深度克隆呢？简单地解释一下。

- 浅度克隆：只负责克隆按值传递的数据（比如基本数据类型、String 类型）。
- 深度克隆：除了浅度克隆要克隆的值外，还负责克隆引用类型的数据，基本上就是被克隆实例所有的属性数据都会被克隆出来。

注意　**深度克隆还有一个特点，如果被克隆的对象里面的属性数据是引用类型，也就是属性的类型也是对象，则需要一直递归地克隆下去。这也意味着，要想深度克隆成功，必须要整个克隆所涉及的对象都要正确实现克隆方法，如果其中有一个没有正确实现克隆，那么就会导致克隆失败。**

在前面的例子中实现的克隆就是典型的浅度克隆。下面来看看如何实现深度克隆。

1. 自己实现原型的深度克隆

（1）要演示深度克隆，需要给订单对象添加一个引用类型的属性，这样实现克隆以后，才能看出深度克隆的效果。

定义一个产品对象，也让它实现克隆的功能。产品对象实现的是一个浅度克隆。

先来定义产品的原型接口。示例代码如下：

```
/**
 * 声明一个克隆产品自身的接口
 */
public interface ProductPrototype {
    /**
     * 克隆产品自身的方法
     * @return 一个从自身克隆出来的产品对象
     */
    public ProductPrototype cloneProduct();
}
```

接下来看看具体的产品对象实现。示例代码如下：

```
/**
 * 产品对象
 */
public class Product implements ProductPrototype{
    /**
     * 产品编号
     */
    private String productId;
    /**
     * 产品名称
     */
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
```

```
        this.name = name;
    }
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
    public String toString(){
        return "产品编号="+this.productId+"，产品名称="+this.name;
    }
    public ProductPrototype cloneProduct() {
        //创建一个新的订单，然后把本实例的数据复制过去
        Product product = new Product();
        product.setProductId(this.productId);
        product.setName(this.name);
        return product;
    }
}
```

（2）订单的具体实现上也需要改变一下，需要在其属性上添加一个产品类型的属性，然后也需要实现克隆方法。示例代码如下：

```
public class PersonalOrder implements OrderApi{
    private String customerName;
    private int orderProductNum = 0;
    /**
     * 产品对象
    */
    private Product product = null;
    public int getOrderProductNum() {
        return this.orderProductNum;
    }
    public void setOrderProductNum(int num) {
        this.orderProductNum = num;
    }
    public String getCustomerName() {
        return customerName;
    }
    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
```

```
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
    public String toString(){
        //简单点输出
        return "订购产品是="+this.product.getName()
                          +"，订购数量为="+this.orderProductNum;
    }
    public OrderApi cloneOrder() {
        //创建一个新的订单，然后把本实例的数据复制过去
        PersonalOrder order = new PersonalOrder();
        order.setCustomerName(this.customerName);
        order.setOrderProductNum(this.orderProductNum);
        //对于对象类型的数据，深度克隆的时候需要继续调用这个对象的克隆方法
        order.setProduct((Product)this.product.cloneProduct());
        return order;
    }
}
```

注意这句话，体现了深度克隆

（3）编写客户端程序测试一下，是否深度克隆成功。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //先创建原型实例
        PersonalOrder oa1 = new PersonalOrder();
        //设置原型实例的值
        Product product = new Product();
        product.setName("产品1");
        oa1.setProduct(product);
        oa1.setOrderProductNum(100);

        System.out.println("这是第一次获取的对象实例="+oa1);

        //通过克隆来获取新的实例
        PersonalOrder oa2 = (PersonalOrder)oa1.cloneOrder();
        //修改它的值
        oa2.getProduct().setName("产品2");
```

```
        oa2.setOrderProductNum(80);
        //输出克隆出来的对象的值
        System.out.println("输出克隆出来的实例="+oa2);

        //再次输出原型实例的值
        System.out.println("再次输出原型实例="+oa1);
    }
}
```

（4）运行结果如下。很明显，我们自己做的深度克隆是成功的。

```
这是第一次获取的对象实例=订购产品是=产品1，订购数量为=100
输出克隆出来的实例=订购产品是=产品2，订购数量为=80
再次输出原型实例=订购产品是=产品1，订购数量为=100
```

（5）小结。

看来自己实现深度克隆也不是很复杂，但是比较麻烦，如果产品类中又有属性是引用类型，在产品类实现克隆方法的时候，则需要调用那个引用类型的克隆方法了。这样一层一层调地下去，如果中途有任何一个对象没有正确实现深度克隆，那将会引起错误，这也是深度克隆容易出错的原因。

1. Java 中的深度克隆

利用 Java 中的 clone 方法来实现深度克隆，大体上和自己做差不多，但是也有一些需要注意的地方，一起来看看吧。

（1）产品类没有太大的不同，主要是把实现的接口变成了 Cloneable，这样一来，实现克隆的方法就不是 cloneProduct，而是变成 clone 方法了；另外一个是克隆方法的实现变成使用“super.clone();”了。示例代码如下：

```
public class Product implements Cloneable{
    private String productId;
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
```

第一个变化：不再实现自己的接口了

```
    public String toString(){
        return "产品编号="+this.productId+", 产品名称="+this.name;
    }
    public Object clone() {
        Object obj = null;
        try {
            obj = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return obj;
    }
}
```

重要变化：不再自己一个一个属性地赋值了，直接调用 super.clone()

（2）具体的订单实现类，除了改变接口外，更重要的是在实现 clone 方法的时候，不仅调用“super.clone();”，还必须显式地调用引用类型属性的 clone 方法，也就是产品的 clone 方法。示例代码如下：

```
public class PersonalOrder implements Cloneable , OrderApi{
    private String customerName;
    private Product product = null;
    private int orderProductNum = 0;
    public int getOrderProductNum() {
        return this.orderProductNum;
    }
    public void setOrderProductNum(int num) {
        this.orderProductNum = num;
    }
    public String getCustomerName() {
        return customerName;
    }
    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
    public String toString(){
        //简单点输出
```

实现的接口发生了改变

```
        return "订购产品是="+this.product.getName()
                            +",订购数量为="+this.orderProductNum;
    }
    public Object clone(){
        PersonalOrder obj=null;
        try {
            obj =(PersonalOrder)super.clone();
            //下面这一句话不可少
            obj.setProduct(
                (Product)this.product.clone());
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return obj;
    }
}
```

重要改变：新的 clone 方法的实现

（3）特别强调。

不可缺少"obj.setProduct((Product)this.product.clone());"这句话。为什么呢？

原因在于调用 super.clone()方法的时候，Java 是先开辟一块内存的空间，然后把实例对象的值原样拷贝过去，对于基本数据类型这样做是没有问题的，而属性 product 是一个引用类型，把值拷贝过去的意思就是把对应的内存地址拷贝过去了，也就是说克隆后的对象实例的 product 和原型对象实例的 product 指向的是同一块内存空间，是同一个产品实例。

因此要想正确地执行深度拷贝，必须手工地对每一个引用类型的属性进行克隆，并重新设置，覆盖掉 super.clone()所拷贝的值。

（4）客户端测试类跟刚才自己做的深度拷贝差不多，只是调用克隆的方法，原来是调用的 cloneOrder 方法，现在变成调用 clone()。运行测试，结果如下：

```
这是第一次获取的对象实例=订购产品是=产品1，订购数量为=100
输出克隆出来的实例=订购产品是=产品2，订购数量为=80
再次输出原型实例=订购产品是=产品1，订购数量为=100
```

注意观察上面的数据，很明显这是正确的，修改克隆出来的实例的属性值，不会影响到原对象实例的属性值。

（5）下面去掉"obj.setProduct((Product)this.product.clone());"这句话，看看会发生什么，运行结果如下：

```
这是第一次获取的对象实例=订购产品是=产品1，订购数量为=100
输出克隆出来的实例=订购产品是=产品2，订购数量为=80
再次输出原型实例=订购产品是=产品2，订购数量为=100
```

仔细观察一下，尤其是加粗的两行，你就会发现，修改克隆对象实例的产品名称属

性的值，影响了原型对象实例的值，这说明没有正确深度克隆。

9.3.4 原型管理器

如果一个系统中原型的数目不固定，比如系统中的原型可以被动态地创建和销毁，那么就需要在系统中维护一个当前可用的原型的注册表，这个注册表就被称为原型管理器。

其实如果把原型当成一个资源的话，原型管理器就相当于一个资源管理器，在系统开始运行的时候初始化，然后运行期间可以动态地添加和销毁资源。从这个角度看，原型管理器就可以相当于一个缓存资源的实现，只不过里面缓存和管理的是原型实例而已。

有了原型管理器后，一般情况下，除了向原型管理器里面添加原型对象的时候是通过 new 来创造的对象，其余时候都是通过向原型管理器来请求原型实例，然后通过克隆方法来获取新的对象实例，这就可以实现动态管理，或者动态切换具体的实现对象实例。

还是通过示例来说明如何实现原型管理器。

（1）先定义原型的接口。非常简单，除了克隆方法，还提供一个名称的属性。示例代码如下：

```
public interface Prototype {
    public Prototype clone();
    public String getName();
    public void setName(String name);
}
```

（2）接下来看看两个具体的实现，实现方式基本上是一样的。

先看第一个原型的实现。示例代码如下：

```
public class ConcretePrototype1 implements Prototype {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Prototype clone() {
        ConcretePrototype1 prototype = new ConcretePrototype1();
        prototype.setName(this.name);
        return prototype;
    }
    public String toString(){
        return "Now in Prototype1, name="+name;
    }
}
```

再看看第二个原型的实现。示例代码如下：

```
public class ConcretePrototype2 implements Prototype {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Prototype clone() {
        ConcretePrototype2 prototype = new ConcretePrototype2();
        prototype.setName(this.name);
        return prototype;
    }
    public String toString(){
        return "Now in Prototype2, name="+name;
    }
}
```

（3）下面来看看原型管理器的实现示意。示例代码如下：

```
/**
 * 原型管理器
 */
public class PrototypeManager {
    /**
     * 用来记录原型的编号和原型实例的对应关系
    */
    private static Map<String,Prototype> map =
                                new HashMap<String,Prototype>();
    /**
     * 私有化构造方法，避免外部无谓的创建实例
     */
    private PrototypeManager(){
        //
    }
    /**
    * 向原型管理器里面添加或是修改某个原型注册
    * @param prototypeId 原型编号
    * @param prototype 原型实例
    */
```

```
    public synchronized static void setPrototype(
                             String prototypeId,Prototype prototype){
        map.put(prototypeId, prototype);
    }
    /**
    * 从原型管理器里面删除某个原型注册
    * @param prototypeId 原型编号
     */
    public synchronized static void removePrototype(
                               String prototypeId){
        map.remove(prototypeId);
    }
    /**
     * 获取某个原型编号对应的原型实例
     * @param prototypeId 原型编号
     * @return 原型编号对应的原型实例
     * @throws Exception 如果原型编号对应的原型实例不存在，报出例外
     */
    public synchronized static Prototype getPrototype(
                             String prototypeId)throws Exception{
        Prototype prototype = map.get(prototypeId);
        if(prototype == null){
            throw new Exception("您希望获取的原型还没有注册或已被销毁");
        }
        return prototype;
    }
}
```

大家会发现，原型管理器是类似一个工具类的实现方式，而且对外的几个方法都是加了同步的，这主要是因为如果在多线程环境下使用这个原型管理器的话，那个 map 属性很明显就成了大家竞争的资源，因此需要加上同步。

（4）下面来看看客户端如何使用这个原型管理器。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        try {
            // 初始化原型管理器
            Prototype p1 = new ConcretePrototype1();
            PrototypeManager.setPrototype("Prototype1", p1);

            // 获取原型来创建对象
```

```
                Prototype p3 = PrototypeManager
                                .getPrototype("Prototype1").clone();
                p3.setName("张三");
                System.out.println("第一个实例: " + p3);

                // 有人动态地切换了实现
                Prototype p2 = new ConcretePrototype2();
                PrototypeManager.setPrototype("Prototype1", p2);

                // 重新获取原型来创建对象
                Prototype p4 = PrototypeManager
                                .getPrototype("Prototype1").clone();
                p4.setName("李四");
                System.out.println("第二个实例: " + p4);

                // 有人注销了这个原型
                PrototypeManager.removePrototype("Prototype1");

                // 再次获取原型来创建对象
                Prototype p5 = PrototypeManager
                                  .getPrototype("Prototype1").clone();
                p5.setName("王五");
                System.out.println("第三个实例: " + p5);
            } catch (Exception err) {
                System.err.println(err.getMessage());
            }
        }
}
```

运行一下，看看结果。示例如下：

```
第一个实例: Now in Prototype1, name=张三
第二个实例: Now in Prototype2, name=李四
您希望获取的原型还没有注册或已被销毁
```

9.3.5 原型模式的优缺点

原型模式的优点

- 对客户端隐藏具体的实现类型
  原型模式的客户端只知道原型接口的类型，并不知道具体的实现类型，从而减少了客户端对这些具体实现类型的依赖。

- 在运行时动态改变具体的实现类型
  原型模式可以在运行期间，由客户来注册符合原型接口的实现类型，也可以动态地改变具体的实现类型，看起来接口没有任何变化，但其实运行的已经是另外一个类实例了。因为克隆一个原型就类似于实例化一个类。

原型模式的缺点

原型模式最大的缺点就在于每个原型的子类都必须实现 clone 的操作，尤其在包含引用类型的对象时，clone 方法会比较麻烦，必须要能够递归地让所有的相关对象都要正确地实现克隆。

9.3.6 思考原型模式

1. 原型模式的本质

> 原型模式的本质：**克隆生成对象。**

克隆是手段，目的是生成新的对象实例。正是因为原型的目的是为了生成新的对象实例，原型模式通常是被归类为创建型的模式。

原型模式也可以用来解决“只知接口而不知实现的问题”，使用原型模式，可以出现一种独特的“接口造接口”的景象，这在面向接口编程中很有用。同样的功能也可以考虑使用工厂来实现。

另外，原型模式的重心还是在创建新的对象实例，至于创建出来的对象，其属性的值是否一定要和原型对象属性的值完全一样，这个并没有强制规定，只不过在目前大多数实现中，克隆出来的对象和原型对象的属性值是一样的。

也就是说，可以通过克隆来创造值不一样的实例，但是对象类型必须一样。可以有部分甚至是全部的属性的值不一样，可以有选择性地克隆，就当是标准原型模式的一个变形使用吧。

2. 何时选用原型模式

建议在以下情况时选用原型模式。

- 如果一个系统想要独立于它想要使用的对象时，可以使用原型模式，让系统只面向接口编程，在系统需要新的对象的时候，可以通过克隆原型来得到。
- 如果需要实例化的类是在运行时刻动态指定时，可以使用原型模式，通过克隆原型来得到需要的实例。

9.3.7 相关模式

- 原型模式和抽象工厂模式
  功能上有些相似，都是用来获取一个新的对象实例的。

不同之处在于，原型模式的着眼点是在如何创造出实例对象来，最后选择的方案是通过克隆；而抽象工厂模式的着眼点则在于如何来创造产品簇，至于具体如何创建出产品簇中的每个对象实例，抽象工厂模式则不是很关注。

正是因为它们的关注点不一样，所以它们也可以配合使用，比如在抽象工厂模式里面，具体创建每一种产品的时候就可以使用该种产品的原型，也就是抽象工厂管产品簇，具体的每种产品怎么创建则可以选择原型模式。

- 原型模式和生成器模式

  这两种模式可以配合使用。

  生成器模式关注的是构建的过程，而在构建的过程中，很可能需要某个部件的实例，那么很自然地就可以应用上原型模式，通过原型模式来得到部件的实例。

第 10 章　中介者模式（Mediator）

10.1 场景问题

10.1.1 如果没有主板

大家都知道，电脑里面各个配件之间的交互，主要是通过主板来完成的（事实上主板有很多的功能，这里不去讨论）。试想一下，如果电脑里面没有主板，会怎样呢？

如果电脑里面没有了主板，那么各个配件之间就必须自行相互交互，以互相传送数据。理论上说，基本上各个配件相互之间都存在交互数据的可能，如图 10.1 所示。

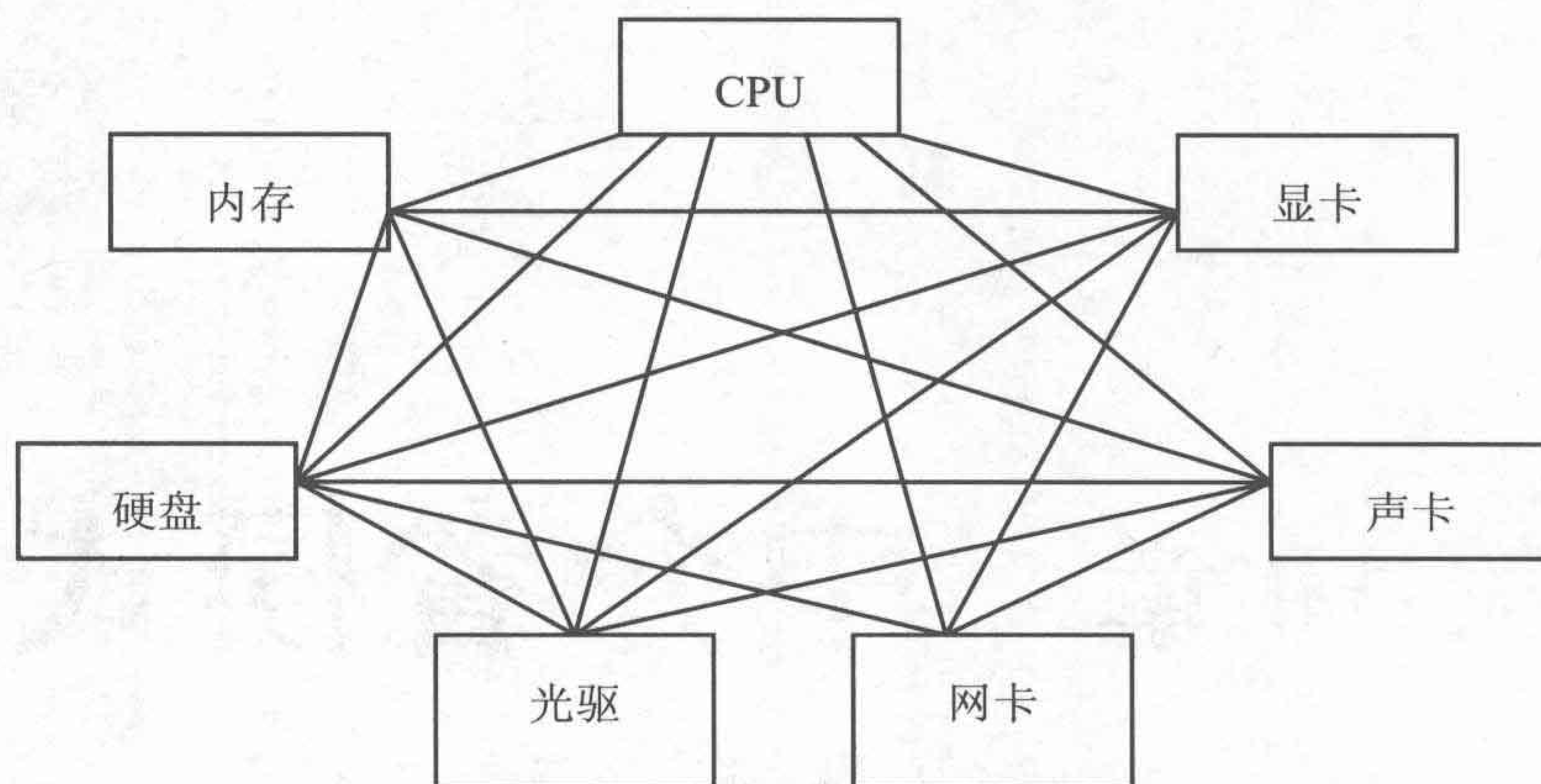


图 10.1 没有主板，各个配件相互交互示意图

这也太复杂了吧，这还没完呢，由于各个配件的接口不同，那么相互之间交互的时候，还必须把数据接口进行转换才能匹配上，那就更恐怖了。

所幸是有了主板，各个配件的交互完全通过主板来完成，每个配件都只需要和主板交互，而主板知道如何和所有的配件打交道，那就简单多了，这就避免了如图 10.1 所描述的那样乱作一团。有主板后的结构如图 10.2 所示：



图 10.2 有主板后的结构示意图

10.1.2 有何问题

如果上面的情况发生在软件开发上呢？

若把每个电脑配件都抽象成为一个类或者是子系统，那就相当于出现了多个类之间相互交互，而且交互很繁琐，导致每个类都必须知道所有需要交互的类，也就是我们常说的类和类耦合了，是不是很麻烦？

在软件开发中出现这种情况可就不妙了，不但开发的时候每个类会复杂，因为要兼顾其他的类，更要命的是每个类在发生改动的时候，需要通知所有相关的类一起修改，因为接口或者是功能发生了变动，使用它的地方都得变，快要疯了吧！

那该如何来简化这种多个对象之间的交互呢？

10.1.3　使用电脑来看电影

为了演示，考虑一个稍微具体点的功能。在日常生活中，我们经常使用电脑来看电影，把这个过程描述出来，这里仅仅考虑正常的情况，也就是有主板的情况，简化后假定会有如下的交互过程。

- 首先是光驱要读取光盘上的数据，然后告诉主板，它的状态改变了。
- 主板去得到光驱的数据，把这些数据交给 CPU 进行分析处理。
- CPU 处理完后，把数据分成了视频数据和音频数据，通知主板，它处理完了。
- 主板去得到 CPU 处理过后的数据，分别把数据交给显卡和声卡，去显示出视频和发出声音。

当然这是一个持续的、不断重复的过程，从而形成不间断的视频和声音。具体的运行过程不在讨论之列，假设就有如上简单的交互关系就可以了。也就是说想看电影，把光盘放入光驱，光驱开始读盘，就可以看电影了。

现在要求使用程序把这个过程描述出来，该如何具体实现呢？

10.2　解决方案

10.2.1　使用中介者模式来解决问题

用来解决上述问题的一个合理的解决方案就是中介者模式（Mediator）。那么什么是中介者模式呢？

1. 中介者模式的定义

> 用一个中介对象来封装一系列的对象交互。中介者使得各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

2. 应用中介者模式来解决问题的思路

仔细分析上面的问题，根本原因就在于多个对象需要相互交互，从而导致对象之间紧密耦合，不利于对象的修改和维护。

中介者模式的解决思路很简单，跟电脑的例子一样，中介者模式通过引入一个中介对象，让其他的对象都只和中介对象交互，而中介对象知道如何和其他所有的对象交互，

这样对象之间的交互关系就没有了，从而实现了对象之间的解耦。

对于中介对象而言，所有相互交互的对象，被视为同事类，中介对象就是来维护各个同事之间的关系，而所有的同事类都只是和中介对象交互。

每个同事对象，当自己发生变化的时候，不需要知道这会引起其他对象有什么变化，它只需要通知中介者就可以了，然后由中介者去与其他对象交互。这样松散耦合带来的好处是，除了让同事对象之间相互没有关联外，还有利于功能的修改和扩展。

有了中介者以后，所有的交互都封装到中介者对象里面，各个对象就不再需要维护这些关系了。扩展关系的时候也只需要扩展或修改中介者对象就可以了。

10.2.2 中介者模式的结构和说明

中介者模式的结构如图 10.3 所示：

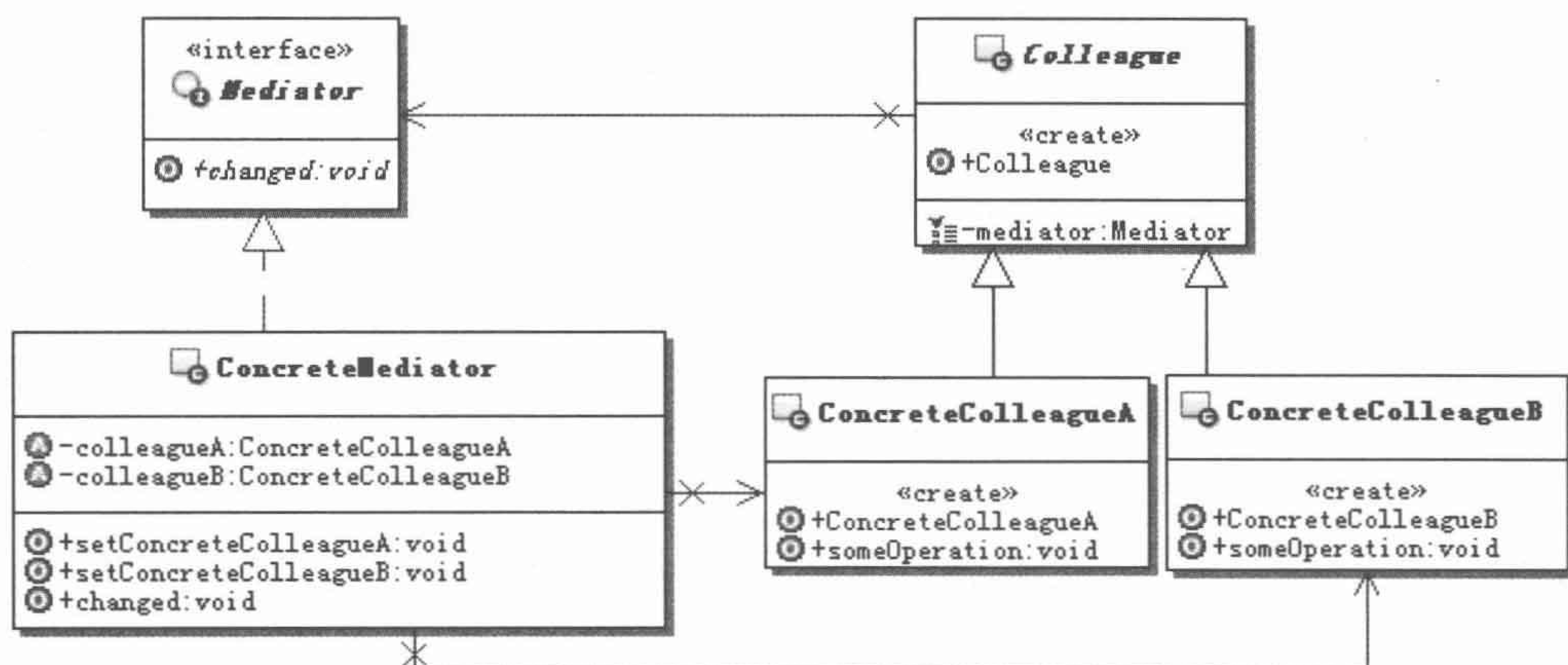


图 10.3 中介者模式结构示意图

- Mediator：中介者接口。在里面定义各个同事之间交互需要的方法，可以是公共的通信方法，比如 changed 方法，大家都用，也可以是小范围的交互方法。
- ConcreteMediator：具体中介者实现对象。它需要了解并维护各个同事对象，并负责具体的协调各同事对象的交互关系。
- Colleague：同事类的定义，通常实现成为抽象类，主要负责约束同事对象的类型，并实现一些具体同事类之间的公共功能，比如，每个具体同事类都应该知道中介者对象，也就是具体同事类都会持有中介者对象，都可以定义到这个类里面。
- ConcreteColleague：具体的同事类，实现自己的业务，在需要与其他同事通信的时候，就与持有的中介者通信，中介者会负责与其他的同事交互。

10.2.3 中介者模式示例代码

（1）先来看看所有同事的父类的定义。

按照前面的描述，所有需要交互的对象都被视为同事类，这些同事类应该有一个统一的约束。而且所有的同事类都需要和中介者对象交互，换句话说就是所有的同事都应

该持有中介者对象。

因此，为了统一约束众多的同事类，并为同事类提供持有中介者对象的公共功能，先来定义一个抽象的同事类，在里面实现持有中介者对象的公共功能。

注意	**要提醒一点，下面示例的这个抽象类是没有定义抽象方法的，主要是用来约束所有同事类的类型。**

示例代码如下：

```
/**
 * 同事类的抽象父类
 */
public abstract class Colleague {
    /**
     * 持有中介者对象，每一个同事类都知道它的中介者对象
     */
    private Mediator mediator;
    /**
     * 构造方法，传入中介者对象
     * @param mediator 中介者对象
     */
    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }
    /**
     * 获取当前同事类对应的中介者对象
     * @return 对应的中介者对象
     */
    public Mediator getMediator() {
        return mediator;
    }
}
```

（2）再来看看具体的同事类，在示意中它们的实现是差不多的。示例代码如下：

```
/**
 * 具体的同事类A
 */
public class ConcreteColleagueA extends Colleague {
    public ConcreteColleagueA(Mediator mediator) {
        super(mediator);
    }
    /**
```

```
     * 示意方法，执行某些业务功能
     */
    public void someOperation() {
        //在需要跟其他同事通信的时候，通知中介者对象
        getMediator().changed(this);
    }
}
```

同事类B的实现。示例代码如下：

```
/**
 * 具体的同事类B
 */
public class ConcreteColleagueB extends Colleague {
    public ConcreteColleagueB(Mediator mediator) {
        super(mediator);
    }
    /**
     * 示意方法，执行某些业务功能
     */
    public void someOperation() {
        //在需要跟其他同事通信的时候，通知中介者对象
        getMediator().changed(this);
    }
}
```

（3）接下来看看中介者的定义。示例代码如下：

```
/**
 * 中介者，定义各个同事对象通信的接口
 */
public interface Mediator {
    /**
     * 同事对象在自身改变的时候来通知中介者的方法
     * 让中介者去负责相应的与其他同事对象的交互
     * @param colleague 同事对象自身，好让中介者对象通过对象实例
     *                  去获取同事对象的状态
     */
    public void changed(Colleague colleague);
}
```

（4）最后来看看具体的中介者实现。示例代码如下：

```
/**
 * 具体的中介者实现
```

```
 */
public class ConcreteMediator implements Mediator {
    /**
     * 持有并维护同事 A
     */
    private ConcreteColleagueA colleagueA;
    /**
     * 持有并维护同事 B
     */
    private ConcreteColleagueB colleagueB;

    /**
     * 设置中介者需要了解并维护的同事 A 对象
     * @param colleague 同事 A 对象
     */
    public void setConcreteColleagueA(
                        ConcreteColleagueA colleague) {
        colleagueA = colleague;
    }
    /**
     * 设置中介者需要了解并维护的同事 B 对象
     * @param colleague 同事 B 对象
     */
    public void setConcreteColleagueB(
                        ConcreteColleagueB colleague) {
        colleagueB = colleague;
    }

    public void changed(Colleague colleague) {
        //某个同事类发生了变化，通常需要与其他同事交互
        //具体协调相应的同事对象来实现协作行为
    }
}
```

10.2.4　使用中介者模式来实现示例

要使用中介者模式来实现示例，那就要区分出同事对象和中介者对象。很明显，主板是作为中介者，而光驱、CPU、声卡、显卡等配件，都是作为同事对象。

根据中介者模式的知识，设计出示例的程序结构，如图 10.4 所示。

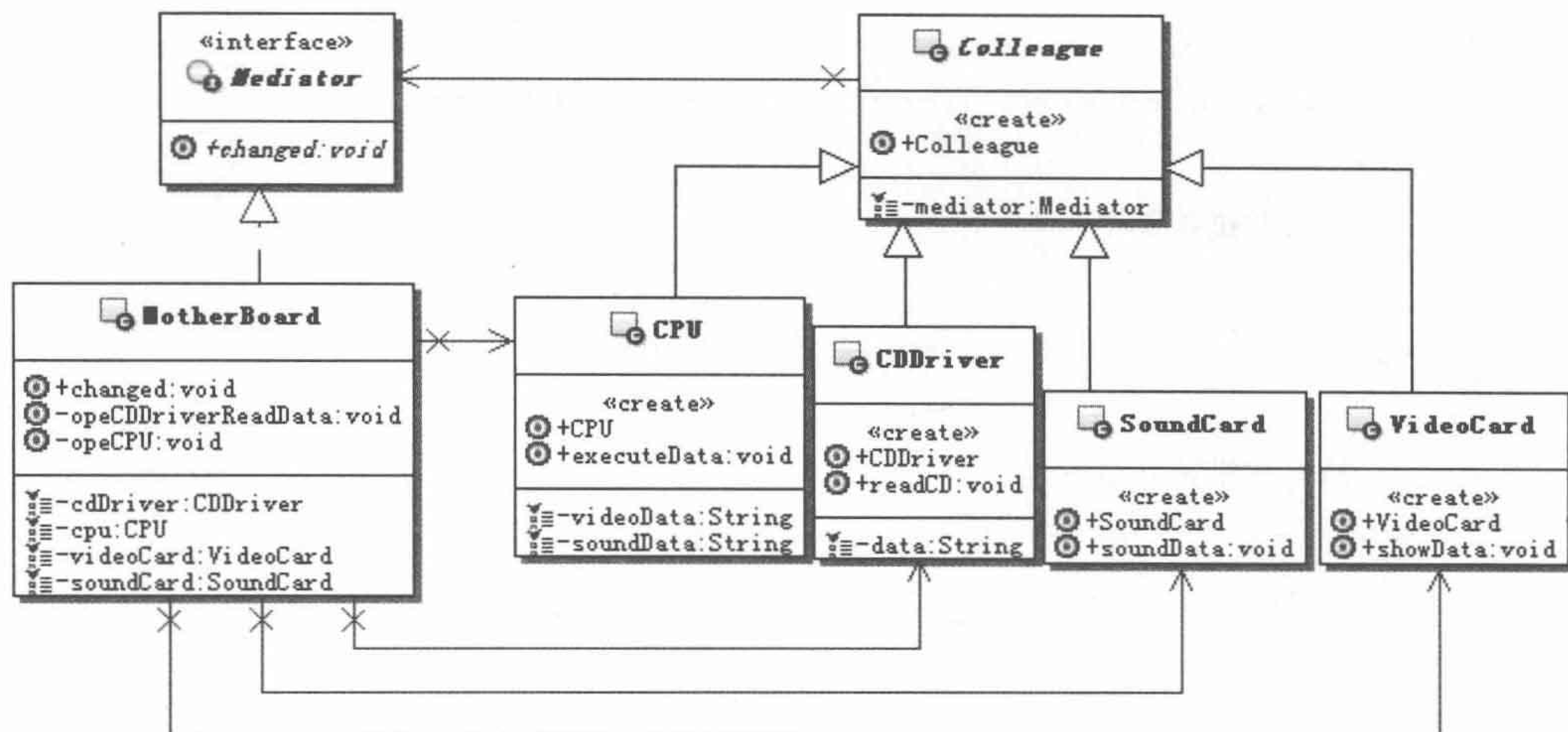


图 10.4　使用中介者模式实现示例的结构示意图

下面来看看代码实现，会更清楚。

（1）先来看看所有同事的抽象父类的定义，跟标准的实现是差不多的。示例代码如下：

```
public abstract class Colleague {
    private Mediator mediator;
    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }
    public Mediator getMediator() {
        return mediator;
    }
}
```

（2）定义众多的同事。

定义好了同事的抽象父类，接下来就应该具体地实现这些同事类了，按照顺序一个一个来。

先看看光驱类吧。示例代码如下：

```
/**
 * 光驱类，一个同事类
 */
public class CDDriver extends Colleague{
    public CDDriver(Mediator mediator) {
        super(mediator);
    }
    /**
     * 光驱读取出来的数据
```

```
     */
    private String data = "";
    /**
     * 获取光驱读取出来的数据
     * @return 光驱读取出来的数据
     */
    public String getData(){
        return this.data;
    }
    /**
     * 读取光盘
     */
    public void readCD(){
        //逗号前是视频显示的数据，逗号后是声音
        this.data = "设计模式,值得好好研究";
        //通知主板，自己的状态发生了改变
        this.getMediator().changed(this);
    }
}
```

自身的业务状态数据

业务方法，也是和中介者交互的方法

接下来该看看 CPU 类了。示例代码如下：

```
/**
 * CPU类，一个同事类
 */
public class CPU extends Colleague{
    public CPU(Mediator mediator) {
        super(mediator);
    }
    /**
     * 分解出来的视频数据
     */
    private String videoData = "";
    /**
     * 分解出来的声音数据
     */
    private String soundData = "";
    /**
     * 获取分解出来的视频数据
     * @return 分解出来的视频数据
     */
    public String getVideoData() {
```

```
        return videoData;
    }
    /**
     * 获取分解出来的声音数据
     * @return 分解出来的声音数据
     */
    public String getSoundData() {
        return soundData;
    }
    /**
     * 处理数据，把数据分成音频和视频的数据
     * @param data 被处理的数据
     */
    public void executeData(String data){
        //把数据分解开，前面的是视频数据，后面的是音频数据
        String [] ss = data.split(",");
        this.videoData = ss[0];
        this.soundData = ss[1];
        //通知主板，CPU 的工作完成
        this.getMediator().changed(this);
    }
}
```


下面该来看看显示的同事类了。显卡类的示例代码如下：

```
/**
 * 显卡类，一个同事类
 */
public class VideoCard extends Colleague{
    public VideoCard(Mediator mediator) {
        super(mediator);
    }
    /**
     * 显示视频数据
     * @param data 被显示的数据
     */
    public void showData(String data){
        System.out.println("您正观看的是："+data);
    }
}
```

同样的，看看声卡的处理类，示例代码如下：

```
/**
 * 声卡类，一个同事类
 */
public class SoundCard extends Colleague{
    public SoundCard(Mediator mediator) {
        super(mediator);
    }
    /**
     * 按照声频数据发出声音
     * @param data 发出声音的数据
     */
    public void soundData(String data){
        System.out.println("画外音："+data);
    }
}
```

（3）定义中介者接口。

由于所有的同事对象都要和中介者交互，下面来定义出中介者的接口，功能不多，提供一个让同事对象在自身改变的时候来通知中介者的方法。示例代码如下：

```
/**
 * 中介者对象的接口
 */
public interface Mediator {
    /**
     * 同事对象在自身改变的时候来通知中介者的方法，
     * 让中介者去负责相应的与其他同事对象的交互
     * @param colleague 同事对象自身，好让中介者对象通过对象实例
     *                  去获取同事对象的状态
     */
    public void changed(Colleague colleague);
}
```

（4）实现中介者对象。

中介者的功能稍微多一点，它需要处理所有的同事对象之间的交互。好在我们要示例的东西并不麻烦，仅有两个功能处理而已。示例代码如下：

```
/**
 * 主板类，实现中介者接口
 */
public class MotherBoard implements Mediator{
    /**
     * 需要知道要交互的同事类——光驱类
```

```
 */
private CDDriver cdDriver = null;
/**
 * 需要知道要交互的同事类——CPU 类
 */
private CPU cpu = null;
/**
 * 需要知道要交互的同事类——显卡类
 */
private VideoCard videoCard = null;
/**
 * 需要知道要交互的同事类——声卡类
 */
private SoundCard soundCard = null;

public void setCdDriver(CDDriver cdDriver) {
    this.cdDriver = cdDriver;
}
public void setCpu(CPU cpu) {
    this.cpu = cpu;
}
public void setVideoCard(VideoCard videoCard) {
    this.videoCard = videoCard;
}
public void setSoundCard(SoundCard soundCard) {
    this.soundCard = soundCard;
}
public void changed(Colleague colleague) {
    if(colleague == cdDriver){
        //表示光驱读取数据了
        this.opeCDDriverReadData((CDDriver)colleague);
    }else if(colleague == cpu){
        //表示 CPU 处理完了
        this.opeCPU((CPU)colleague);
    }
}
/**
 * 处理光驱读取数据以后与其他对象的交互
 * @param cd 光驱同事对象
 */
```

```
    private void opeCDDriverReadData(CDDriver cd){
        //1：先获取光驱读取的数据
        String data = cd.getData();
        //2：把这些数据传递给 CPU 进行处理
        this.cpu.executeData(data);
    }
    /**
     * 处理 CPU 处理完数据后与其他对象的交互
     * @param cpu CPU 同事类
     */
    private void opeCPU(CPU cpu){
        //1：先获取 CPU 处理后的数据
        String videoData = cpu.getVideoData();
        String soundData = cpu.getSoundData();
        //2：把这些数据传递给显卡和声卡展示出来
        this.videoCard.showData(videoData);
        this.soundCard.soundData(soundData);
    }
}
```

（5）看个电影享受一下。

定义完了同事类，也实现处理了它们交互的中介者对象，该来写个客户端使用它们。来看个电影，好好享受一下。写个客户端测试一下，看看效果。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //1：创建中介者——主板对象
        MotherBoard mediator = new MotherBoard();
        //2：创建同事类
        CDDriver cd = new CDDriver(mediator);
        CPU cpu = new CPU(mediator);
        VideoCard vc = new VideoCard(mediator);
        SoundCard sc = new SoundCard(mediator);

        //3：让中介者知道所有的同事
        mediator.setCdDriver(cd);
        mediator.setCpu(cpu);
        mediator.setVideoCard(vc);
        mediator.setSoundCard(sc);

        //4：开始看电影，把光盘放入光驱，光驱开始读盘
```

```
            cd.readCD();
        }
}
```

测试结果如下：

```
您正观看的是：设计模式
画外音：值得好好研究
```

如同上面的示例，对于光驱对象、CPU 对象、显卡对象和声卡对象，需要相互交互，虽然只是简单演示，但是也能看出来，它们的交互是比较麻烦的，于是定义一个中介者对象——主板对象，来维护它们之间的交互关系，从而使得这些对象松散耦合。

如果这个时候需要修改它们的交互关系，直接到中介者里面修改就好了，也就是说它们的关系已经独立封装到中介者对象里面了，可以独立地改变它们之间的交互关系，而不用去修改这些同事对象。

10.3 模式讲解

10.3.1 认识中介者模式

1. 中介者模式的功能

中介者的功能非常简单，就是封装对象之间的交互。如果一个对象的操作会引起其他相关对象的变化，或者是某个操作需要引起其他对象的后续或连带操作，而这个对象又不希望自己来处理这些关系，那么就可以找中介者，把所有的麻烦扔给它，只在需要的时候通知中介者，其他的就让中介者去处理就可以了。

反过来，其他的对象在操作的时候，可能会引起这个对象的变化，也可以这么做。最后对象之间就完全分离了，谁都不直接跟其他对象交互，那么相互的关系全部被集中到中介者对象里面了，所有的对象就只是跟中介者对象进行通信，相互之间不再有联系。

把所有对象之间的交互都封装在中介者当中，无形中还可以得到另外一个好处，就是能够集中地控制这些对象的交互关系，这样当有变化的时候，修改起来就很方便。

2. 需要 Mediator 接口吗

要回答这个问题，先要搞清楚一件事情，接口是用来干什么的？接口是用来实现"封装隔离"的，那么封装谁？隔离谁呢？Mediator 接口就是用来封装中介者对象的，使得使用中介者对象的客户对象跟具体的中介者实现对象分离开。

了解了上面的这些内容，回过来想想，有没有使用 Mediator 接口的必要，那就取决于是否会提供多个不同的中介者实现。如果中介者实现只有一个的话，而且预计中也没有需要扩展的要求，那么就可以不定义 Mediator 接口，让各个同事对象直接使用中介者实现对象；如果中介者实现不只一个，或者预计中有扩展的要求，那么就需要定义 Mediator 接口，让各个同事对象来面向中介者接口编程，而无须关心具体的中介者实现。

3. 同事关系

在标准的中介者模式中，将使用中介者对象来交互的那些对象称为同事类，这不是随便叫的，在中介者模式中，要求这些类都要继承相同的类。也就是说，这些对象从某个角度讲是同一个类型，算是兄弟对象。

正是这些兄弟对象之间的交互关系很复杂，才产生了把这些交互关系分离出去，单独做成中介者对象，这样一来，这些兄弟对象就成了中介者对象眼里的同事。

4. 同事和中介者的关系

在中介者模式中，当一个同事对象发生了改变，需要主动通知中介者，让中介者去处理与其他同事对象相关的交互。

这就导致了同事对象和中介者对象之间必须有关系，首先是同事对象需要知道中介者对象是谁；反过来，中介者对象也需要知道相关的同事对象，这样它才能与同事对象进行交互。也就是说中介者对象和同事对象之间是相互依赖的。

5. 如何实现同事和中介者的通信

一个同事对象发生了改变，会通知中介者对象，中介者对象会处理与其他同事的交互，这就产生了同事对象和中介者对象的相互通信。怎么实现这种通信关系呢？

一种实现方式是在 Mediator 接口中定义一个特殊的通知接口，作为一个通用的方法，让各个同事类来调用这个方法，在中介者模式结构图里画的就是这种方式。在前面示例的也是这种方式，定义了一个通用的 changed 方法，并且把同事对象当做参数传入，这样在中介者对象里面，就可以去获取这个同事对象的实例的数据了。

另外一种实现方式是可以采用观察者模式，把 Mediator 实现成为观察者，而各个同事类实现成为 Subject，这样同事类发生了改变，会通知 Mediator。Mediator 在接到通知以后，会与相应的同事对象进行交互。

6. 中介者模式的调用顺序示意图

中介者模式的调用顺序如图 10.5 所示。

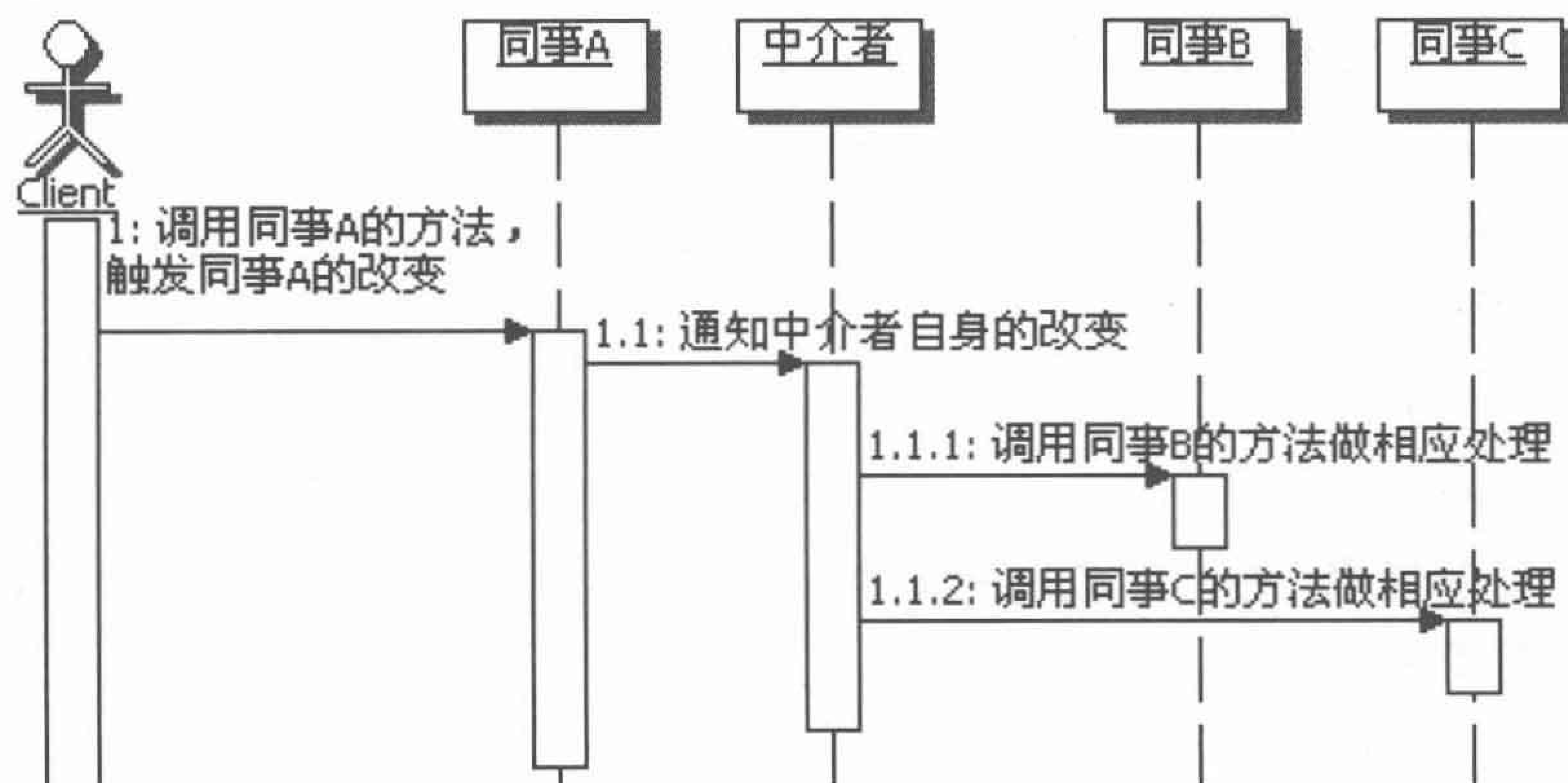


图 10.5　中介者模式的调用顺序示意图

10.3.2 广义中介者

仔细查看中介者的结构、定义和示例，会发现几个问题，使得中介者模式在实际使用的时候，变得繁琐或困难。

其一，是否有必要为同事对象定义一个公共的父类？

大家都知道，Java 是单继承的，为了使用中介者模式，就让这些同事对象继承一个父类，这是很不好的；再说了，这个父类目前也没有什么特别的公共功能，也就是说继承它也得不到多少好处。

在实际开发中，很多相互交互的对象本身是没有公共父类的，强行加上一个父类，会让这些对象实现起来特别别扭。

其二，同事类有必要持有中介者对象吗？

同事类需要知道中介者对象，以便当它们发生改变的时候能够通知中介者对象。但是是否需要作为属性并通过构造方法传入这么强的依赖关系呢？

也可以用简单的方式去通知中介对象，比如把中介对象做成单例，直接在同事类的方法里面去调用中介者对象。

其三，是否需要中介者接口？

在实际开发中，很常见的情况是不需要中介者接口的，而且中介者对象也不需要创建很多个实例。因为中介者是用来封装和处理同事对象的关系的，它一般是没有状态需要维护的，因此中介者通常可以实现成单例。

其四，中介者对象是否需要持有所有的同事？

虽说中介者对象需要知道所有的同事类，这样中介者才能与它们交互。但是是否需要作为属性这么强烈的依赖关系，而且中介者对象在不同的关系维护上，可能会需要不同的同事对象的实例，因此可以在中介者处理的方法里面去创建，或者获取，或者从参数传入需要的同事对象。

其五，中介者对象只是提供一个公共的方法来接受同事对象的通知吗？

从示例中可以看出来，在公共方法里，还是要去区分到底是谁调过来，这还是简单的，还没有去区分到底是什么样的业务触发调用过来的，因为不同的业务，引起的与其他对象的交互是不一样的。

因此在实际开发中，通常会提供具体的业务通知方法，这样就不用再去判断到底是什么对象，具体是什么业务了。

基于上面的考虑，在实际应用开发中，经常会简化中介者模式，来使开发变得简单，比如有如下的简化。

- 通常会去掉同事对象的父类，这样可以让任意的对象，只要需要相互交互，就可以成为同事。
- 通常不定义 Mediator 接口，把具体的中介者对象实现成为单例。
- 同事对象不再持有中介者，而是在需要的时候直接获取中介者对象并调用；中介者也不再持有同事对象，而是在具体处理方法里面去创建，或者获取，或者从参

数传入需要的同事对象。

把这样经过简化、变形使用的情况称为广义中介者。

还是举个实际点的例子来看看吧。

1. 部门与人员

几乎在每个应用系统中都需要这样的功能模块：部门管理和人员管理，为了简单点演示，把模块简化成类，也就是有一个部门类 Dep 和人员类 User。

首先想想部门类 Dep 和人员类 User 之间是什么关系，一对一？一对多？还是多对多？

可能在不同的系统里面，根据需要会做成不同的关系。但从实际情况讲，部门和人员应该是多对多的，也就是一个部门可以有多个人，而一个人也可以加入多个部门。

对于一个部门有多个人，估计大家都能理解。而一个人也可以加入多个部门，或许有些朋友就觉得有些问题了，因为在他们做系统的经验上，是一个人只属于一个部门的。

事实上一个人是可以属于多个部门的，比如，某人是开发部的经理，同时也是销售部门的技术总监，为销售部门给客户的解决方案中的技术部分进行把关，同时还可以是客户服务部门的技术顾问，为他们解决客户的技术问题提供指导。

好了，理解了部门和人员是多对多的关系以后，有些朋友可能会做出如下的设计，不就是个多对多吗，类之间的多对多也很容易表达啊，如下：

```
public class Dep {
    private Collection<User> colUser = new ArrayList<User>();
}
public class User {
    private Collection<Dep> colDep = new ArrayList<Dep>();
}
```

很简单，是吧，一个部门有多个人员，一个人员属于多个部门。

2. 问题的出现

真的这么简单吗？再进一步想想部门和人员的功能交互，就会知道这样设计是存在问题的，举几个常见的功能：

- 部门被撤销；
- 部门之间进行合并；
- 人员离职；
- 人员从一个部门调职到另外一个部门。

想想要实现这些功能，按照前面的设计，该怎么做呢？

（1）系统运行期间部门被撤销了，就意味着这个部门不存在了。可是原来这个部门下所有的人员，每个人员的所属部门中都有这个部门呢，那么就需要先通知所有的人员，把这个部门从他们的所属部门中去掉，然后才可以清除这个部门。

（2）部门合并。是合并成一个新的部门呢，还是把一个部门并入到另一个部门？如果是合并成一个新的部门，那么需要把原有的两个部门撤销，然后再新增一个部门；如

果是把一个部门合并到另一个部门里面，那就是撤销掉一个部门，然后把这个部门下的人员移动到这个部门。不管是哪种情况，都面临着需要通知相应的人员进行更改这样的问题。

（3）人员离职了，反过来就需要通知他所属于的部门，从部门的拥有人员的记录中去除这个人员。

（4）人员调职，同样需要通知相关的部门，先从原来的部门中去除掉，然后再到新的部门中添加上。

看了上述的描述，感觉如何？是不是就一个字“烦”啊！

麻烦的根源在什么地方呢？仔细想想，对了，麻烦的根源就在于部门和人员之间的耦合，这样导致操作人员的时候，需要操作所有相关的部门，而操作部门的时候又需要操作所有相关的人员，使得部门和人员搅和在了一起。

3. 中介者来解决

找到了根源就好办了，采用中介者模式，引入一个中介者对象来管理部门和人员之间的关系，就能解决这些问题了。

如果采用标准的中介者模式，想想上面提出的那些问题点吧，就知道实现起来会很别扭。因此采用广义的中介者来解决，这样部门和人员就完全解耦了，也就是说部门不知道人员，人员也不知道部门，它们完全分开，它们之间的关系就完全由中介者对象来管理了。这个时候的结构如图 10.6 所示。

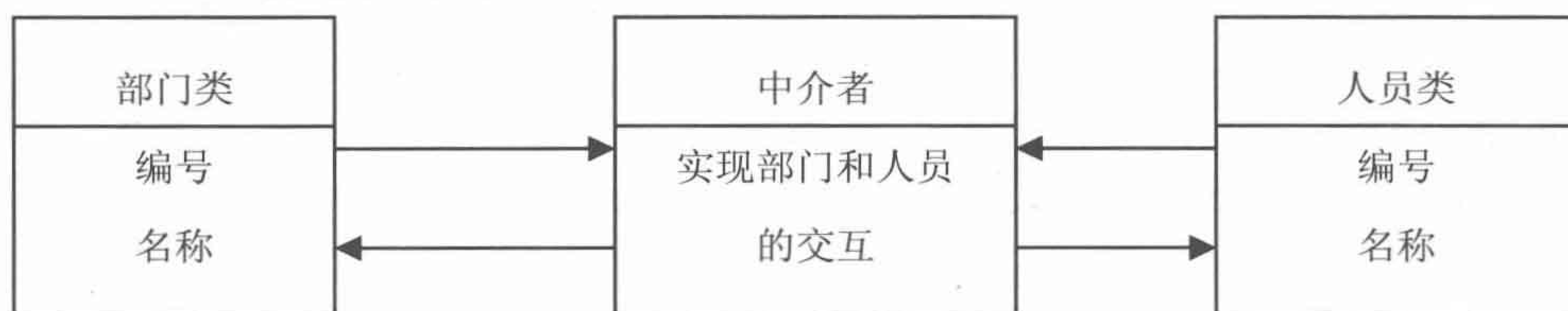


图 10.6　引入中介者后的结构示意图

好了，还是看代码示例会比较清晰。

4. 实现示例

（1）首先定义部门类。示例代码如下：

```
/**
 * 部门类
 */
public class Dep{
    /**
     * 描述部门编号
     */
    private String depId;
    /**
     * 描述部门名称
```

```
     */
    private String depName;

    public String getDepId() {
        return depId;
    }
    public void setDepId(String depId) {
        this.depId = depId;
    }
    public String getDepName() {
        return depName;
    }
    public void setDepName(String depName) {
        this.depName = depName;
    }
    /**
     * 撤销部门
     * @return 是否撤销成功
     */
    public boolean deleteDep(){
        //1：要先通过中介者去除掉所有与这个部门相关的部门和人员的关系
        DepUserMediatorImpl mediator =
                            DepUserMediatorImpl.getInstance();
        mediator.deleteDep(depId);
        //2：然后才能真正地清除掉这个部门
        //请注意在实际开发中，这些业务功能可能会做到业务层去
        //而且实际开发中对于已经使用的业务数据通常是不会被删除的
        //而是会被作为历史数据保留
        return true;
    }
}
```

（2）接下来定义人员类。示例代码如下：

```
/**
 * 人员类
 */
public class User{
    /**
     * 人员编号
     */
    private String userId;
```

```
    /**
     * 人员名称
     */
    private String userName;

    public String getUserId() {
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    /**
     * 人员离职
     * @return 是否处理成功
     */
    public boolean dimission(){
        //1：要先通过中介者去除掉所有与这个人员相关的部门和人员的关系
        DepUserMediatorImpl mediator =
                                DepUserMediatorImpl.getInstance();
        mediator.deleteUser(userId);
        //2：然后才能真正地清除掉这个人员
        //请注意，实际开发中，人员离职，是不会真的删除人员记录的
        //通常是把人员记录的状态或者是删除标记设置成已删除
        //只是不再参加新的业务，但是已经发生的业务记录是不会被清除掉的

        return true;
    }
}
```

（3）顺带看一下描述部门和人员关系的对象，非常简单。示例代码如下：

```
/**
 *  描述部门和人员关系的类
 */
public class DepUserModel {
    /**
```

```
     * 用于部门和人员关系的编号，用做主键
     */
    private String depUserId;
    /**
     * 部门的编号
     */
    private String depId;
    /**
     * 人员的编号
     */
    private String userId;

                      属性对应的 getter/setter 方法，
                      因为篇幅关系，省略了
}
```

（4）具体的中介者实现。

首先中介者要管理部门和人员的关系，所以在中介者实现里面添加了一些测试的数据，为此还专门做了一个用来描述部门和人员关系的数据对象；其次在中介者里面只是实现了撤销部门和人员离职相应的关系处理，其他的没有实现；另外，这个中介者实现被实现成单例的了。示例代码如下：

```
/**
 * 实现部门和人员交互的中介者实现类
 * 说明：为了演示的简洁性，只示例实现撤销部门和人员离职的功能
 */
public class DepUserMediatorImpl{
    private static DepUserMediatorImpl mediator =
new DepUserMediatorImpl();
    private DepUserMediatorImpl(){
        //调用初始化测试数据的功能
        initTestData();
    }                                                   实现成单例
    public static DepUserMediatorImpl getInstance(){
        return mediator;
    }

    /**
     * 测试用，记录部门和人员的关系
     */
    private Collection<DepUserModel> depUserCol =
```

```
                                new ArrayList<DepUserModel>();
    /**
     * 初始化测试数据
     */
    private void initTestData(){
        //准备一些测试数据
        DepUserModel du1 = new DepUserModel();
        du1.setDepUserId("du1");
        du1.setDepId("d1");
        du1.setUserId("u1");
        depUserCol.add(du1);

        DepUserModel du2 = new DepUserModel();
        du2.setDepUserId("du2");
        du2.setDepId("d1");
        du2.setUserId("u2");
        depUserCol.add(du2);

        DepUserModel du3 = new DepUserModel();
        du3.setDepUserId("du3");
        du3.setDepId("d2");
        du3.setUserId("u3");
        depUserCol.add(du3);

        DepUserModel du4 = new DepUserModel();
        du4.setDepUserId("du4");
        du4.setDepId("d2");
        du4.setUserId("u4");
        depUserCol.add(du4);

        DepUserModel du5 = new DepUserModel();
        du5.setDepUserId("du5");
        du5.setDepId("d2");
        du5.setUserId("u1");
        depUserCol.add(du5);
    }
    /**
     * 完成因撤销部门的操作所引起的与人员的交互，需要去除相应的关系
     * @param depId 被撤销的部门对象的编号
     * @return 是否已经正确地处理了因撤销部门所引起的与人员的交互
```

```
     */
    public boolean deleteDep(String depId) {
        //请注意：为了演示简单，部门撤销后，
        //原部门的人员怎么处理等后续业务处理，这里就不管了

        //1：到记录部门和人员关系的集合里面，寻找跟这个部门相关的人员
        //设置一个临时的集合，记录需要清除的关系对象
        Collection<DepUserModel> tempCol =
                                    new ArrayList<DepUserModel>();
        for(DepUserModel du : depUserCol){
            if(du.getDepId().equals(depId)){
                //2：需要把这个相关的记录去掉，先记录下来
                tempCol.add(du);
            }
        }
        //3：从关系集合里面清除掉这些关系
        depUserCol.removeAll(tempCol);

        return true;
    }
    /**
     * 完成因人员离职引起的与部门的交互
     * @param userId 离职人员的编号
     * @return 是否正确处理了因人员离职引起的与部门的交互
     */
    public boolean deleteUser(String userId) {
        //1：到记录部门和人员关系的集合里面，寻找跟这个人员相关的部门
        //设置一个临时的集合，记录需要清除的关系对象
        Collection<DepUserModel> tempCol =
                                    new ArrayList<DepUserModel>();
        for(DepUserModel du : depUserCol){
            if(du.getUserId().equals(userId)){
                //2：需要把这个相关的记录去掉，先记录下来
                tempCol.add(du);
            }
        }
        //3：从关系集合里面清除掉这些关系
        depUserCol.removeAll(tempCol);

        return true;
```

```
}
/**
 * 测试用，在内部打印显示一个部门下的所有人员
 * @param dep 部门对象
 */
public void showDepUsers(Dep dep) {
    for(DepUserModel du : depUserCol){
        if(du.getDepId().equals(dep.getDepId())){
            System.out.println("部门编号="+dep.getDepId()
                    +"下面拥有人员，其编号是："+du.getUserId());
        }
    }
}
/**
 * 测试用，在内部打印显示一个人员所属的部门
 * @param user 人员对象
 */
public void showUserDeps(User user) {
    for(DepUserModel du : depUserCol){
        if(du.getUserId().equals(user.getUserId())){
            System.out.println("人员编号="+user.getUserId()
                    +"属于部门编号是："+du.getDepId());
        }
    }
}
/**
 * 完成因人员调换部门引起的与部门的交互
 * @param userId 被调换的人员的编号
 * @param oldDepId 调换前的部门的编号
 * @param newDepId 调换后的部门的编号
 * @return 是否正确处理了因人员调换部门引起的与部门的交互
 */
public boolean changeDep(String userId,String oldDepId
                        , String newDepId) {
    //本示例不去实现了
    return false;
}
/**
 * 完成因部门合并操作所引起的与人员的交互
 * @param colDepIds 需要合并的部门的编号集合
```

```
     * @param newDep 合并后新的部门对象
     * @return 是否正确处理了因部门合并操作所引起的与人员的交互
     */
    public boolean joinDep(Collection<String> colDepIds
                                        , Dep newDep){
        //本示例不去实现了
        return false;
    }
}
```

（5）测试一下，看看好用不。客户端示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        DepUserMediatorImpl mediator =
                                DepUserMediatorImpl.getInstance();
        //准备要撤销的部门，仅仅需要一个部门编号
        Dep dep = new Dep();
        dep.setDepId("d1");
        Dep dep2 = new Dep();
        dep2.setDepId("d2");
        //准备用于测试的人员，也只需要一个人员编号
        User user = new User();
        user.setUserId("u1");

        //测试撤销部门，在运行之前，输出一下，看这个人员属于哪些部门
        System.out.println("撤销部门前------------------");
        mediator.showUserDeps(user);
        //真正执行业务，撤销这个部门
        dep.deleteDep();
        //再次输出一下，看这个人员属于哪些部门
        System.out.println("撤销部门后------------------");
        mediator.showUserDeps(user);

        //测试人员离职，在运行之前，输出一下，看这个部门下都有哪些人员
        System.out.println("---------------------------------");
        System.out.println("人员离职前------------------");
        mediator.showDepUsers(dep2);
        //真正执行业务，人员离职
        user.dimission();
        //再次输出一下，看这个部门下都有哪些人员
        System.out.println("人员离职后------------------");
```

```
        mediator.showDepUsers(dep2);
    }
}
```

测试结果如下：

```
撤销部门前------------------
人员编号=u1 属于部门编号是：d1
人员编号=u1 属于部门编号是：d2
撤销部门后------------------
人员编号=u1 属于部门编号是：d2
---------------------------------
人员离职前------------------
部门编号=d2 下面拥有人员，其编号是：u3
部门编号=d2 下面拥有人员，其编号是：u4
部门编号=d2 下面拥有人员，其编号是：u1
人员离职后------------------
部门编号=d2 下面拥有人员，其编号是：u3
部门编号=d2 下面拥有人员，其编号是：u4
```

好好体会一下，看看这样做是不是变得更容易了些，而且也实现了中介者想要实现的功能，那就是让同事对象相互分离，由中介对象统一管理它们的交互。

10.3.3 中介者模式的优缺点

中介者模式的优点。

- 松散耦合
  中介者模式通过把多个同事对象之间的交互封装到中介者对象里面，从而使得同事对象之间松散耦合，基本上可以做到互不依赖。这样一来，同事对象就可以独立地变化和复用，而不再像以前那样“牵一发而动全身”了。
- 集中控制交互
  多个同事对象的交互，被封装在中介者对象里面集中管理，使得这些交互行为发生变化的时候，只需要修改中介者对象就可以了，当然如果是已经做好的系统，那就扩展中介者对象，而各个同事类不需要做修改。
- 多对多变成一对多
  没有使用中介者模式的时候，同事对象之间的关系通常是多对多的，引入中介者对象以后，中介者对象和同事对象的关系通常变成了双向的一对多，这会让对象的关系更容易理解和实现。

中介者模式的缺点。

中介者模式的一个潜在缺点是，过度集中化。如果同事对象的交互非常多，而且比较复杂，当这些复杂性全部集中到中介者的时候，会导致中介者对象变得十分复杂，而且难于管理和维护。

10.3.4 思考中介者模式

1. 中介者模式的本质

> 中介者模式的本质：**封装交互。**

中介者模式的目的，就是用来封装多个对象的交互，这些交互的处理多在中介者对象里面实现。因此中介对象的复杂程度，就取决于它封装的交互的复杂程度。

只要是实现封装对象之间的交互功能，就可以应用中介者模式，而不必过于拘泥于中介者模式本身的结构。标准的中介者模式限制很多，导致能完全按照标准使用中介者模式的地方并不是很多，而且多集中在界面实现上。只要本质不变，稍稍变形一下，简化一下，或许能更好地使用中介者模式。

2. 何时选用中介者模式

建议在以下情况时选用中介者模式。

- 如果一组对象之间的通信方式比较复杂，导致相互依赖、结构混乱，可以采用中介者模式，把这些对象相互的交互管理起来，各个对象都只需要和中介者交互，从而使得各个对象松散耦合，结构也更清晰易懂。
- 如果一个对象引用很多的对象，并直接跟这些对象交互，导致难以复用该对象，可以采用中介者模式，把这个对象跟其他对象的交互封装到中介者对象里面，这个对象只需要和中介者对象交互就可以了。

10.3.5 相关模式

- 中介者模式和外观模式

  这两个模式有相似的地方，也存在很大的不同。

  外观模式多用来封装一个子系统内部的多个模块，目的是向子系统外部提供简单易用的接口。也就是说外观模式封装的是子系统外部和子系统内部模块间的交互；而中介者模式是提供多个平等的同事对象之间交互关系的封装，一般是用在内部实现上。

  另外，外观模式是实现单向的交互，是从子系统外部来调用子系统内部，不会反着来；而中介者模式实现的是内部多个模块间多向的交互。
- 中介者模式和观察者模式

  这两个模式可以组合使用。

  中介者模式可以组合使用观察者模式，来实现当同事对象发生改变的时候，通知中介对象，让中介对象去进行与其他相关对象的交互。

读书笔记

第 11 章　代理模式（Proxy）

11.1 场景问题

11.1.1 访问多条数据

考虑这样一个实际应用：要一次性访问多条数据。

这个功能的背景是这样的：在一个 HR（人力资源）应用项目中客户提出，当选择一个部门或是分公司的时候，要把这个部门或者分公司下的所有员工都显示出来，而且不要翻页，方便他们进行业务处理。在显示全部员工的时候，只需要显示名称即可，但是也需要提供如下的功能：在必要的时候可以选择并查看某位员工的详细信息。

客户方是一个集团公司，有些部门或者分公司可能有好几百人，不让翻页，也就是要求一次性地获取这多条数据并展示出来。

该怎么样实现呢？

11.1.2 不用模式的解决方案

不就是要获取某个部门或者某个分公司下的所有员工的信息吗？直接使用 sql 语句从数据库中查询就可以得到。示意性的 sql 大致如下：

```
String sql = "select * from 用户表,部门表 "
        +"where 用户表.depId=部门表.depId "
        +"and 部门表.depId like '"+用户选择查看的depId+"%'";
```

为了方便获取某个部门或者某个分公司下的所有员工的信息，设计部门编号的时候，是按照层级来进行编码的，比如，上一级部门的编码为“01”，那么本级的编码就是“0101”、“0102”…以此类推，下一级的编码就是“010101”、“010102”…。

这种设计方式，从设计上看虽然不够优雅，但是实用。像这种获取某个部门或者某个分公司下的所有员工信息的功能，就不用递归去查找了，直接使用 like，只要找到以该编号开头的所有部门就可以了。

示例涉及到的表有两个，一个是用户表，一个是部门表。两个表需要描述的字段都较多，尤其是用户表，多达好几十个，为了示例简洁，简化后简单的定义如下：

```
DROP TABLE TBL_USER CASCADE CONSTRAINTS ;
DROP TABLE TBL_DEP CASCADE CONSTRAINTS ;
CREATE TABLE TBL_DEP (
    DEPID VARCHAR2(20)  PRIMARY KEY,
    NAME  VARCHAR2(20)
);
CREATE TABLE TBL_USER (
    USERID VARCHAR2(20)  PRIMARY KEY,
    NAME VARCHAR2(20) ,
    DEPID VARCHAR2(20) ,
```

```
    SEX VARCHAR2(10) ,
    CONSTRAINT TBL_USER_FK FOREIGN KEY(DEPID)
    REFERENCES TBL_DEP(DEPID)
);
```

全部采用大写，是基于 Oracle 开发的习惯。再来增加点测试数据，SQL 如下：

```
INSERT INTO TBL_DEP VALUES('01','总公司');
INSERT INTO TBL_DEP VALUES('0101','一分公司');
INSERT INTO TBL_DEP VALUES('0102','二分公司');
INSERT INTO TBL_DEP VALUES('010101','开发部');
INSERT INTO TBL_DEP VALUES('010102','测试部');
INSERT INTO TBL_DEP VALUES('010201','开发部');
INSERT INTO TBL_DEP VALUES('010202','客服部');
INSERT INTO TBL_USER VALUES('user0001','张三1','010101','男');
INSERT INTO TBL_USER VALUES('user0002','张三2','010101','男');
INSERT INTO TBL_USER VALUES('user0003','张三3','010102','男');
INSERT INTO TBL_USER VALUES('user0004','张三4','010201','男');
INSERT INTO TBL_USER VALUES('user0005','张三5','010201','男');
INSERT INTO TBL_USER VALUES('user0006','张三6','010202','男');
COMMIT;
```

准备好了表结构和测试数据，下面来看看具体的实现示例。为了示例的简洁，直接使用 JDBC 来完成。

（1）先来定义描述用户数据的对象。示例代码如下：

```
/**
 * 描述用户数据的对象
 */
public class UserModel {
    /**
     * 用户编号
     */
    private String userId;
    /**
     * 用户姓名
     */
    private String name;
    /**
     * 部门编号
     */
    private String depId;
    /**
```

```
     * 性别
     */
    private String sex;
    public String getUserId() {
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDepId() {
        return depId;
    }
    public void setDepId(String depId) {
        this.depId = depId;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
    public String toString(){
        return "userId="+userId+",name="+name+",depId="
                                    +depId+",sex="+sex+"\n";
    }
}
```

（2）接下来使用 JDBC 来实现要求的功能。示例代码如下：

```
/**
 * 实现示例要求的功能
 */
public class UserManager {
    /**
     * 根据部门编号来获取该部门下的所有人员
     * @param depId 部门编号
```

```
     * @return 该部门下的所有人员
     */
    public Collection<UserModel> getUserByDepId(
                                    String depId)throws Exception{
        Collection<UserModel> col = new ArrayList<UserModel>();
        Connection conn = null;
        try{
            conn = this.getConnection();
            String sql = "select * from tbl_user u,tbl_dep d "
                +"where u.depId=d.depId and d.depId like ?";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, depId+"%");

            ResultSet rs = pstmt.executeQuery();
            while(rs.next()){
                UserModel um = new UserModel();
                um.setUserId(rs.getString("userId"));
                um.setName(rs.getString("name"));
                um.setDepId(rs.getString("depId"));
                um.setSex(rs.getString("sex"));

                col.add(um);
            }
            rs.close();
            pstmt.close();
        }finally{
            conn.close();
        }
        return col;
    }
    /**
     * 获取与数据库的连接
     * @return 数据库连接
     */
    private Connection getConnection() throws Exception {
        Class.forName("你用的数据库对应的JDBC驱动类");
        return DriverManager.getConnection(
                "连接数据库的URL", "用户名", "密码");
    }
}
```

（3）写个客户端来测试看看，是否能满足功能要求。示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception{
        UserManager userManager = new UserManager();
        Collection<UserModel> col =
                                userManager.getUserByDepId("0101");
        System.out.println(col);
    }
}
```

运行结果如下：

```
[userId=user0001,name=张三1,depId=010101,sex=男
, userId=user0002,name=张三2,depId=010101,sex=男
, userId=user0003,name=张三3,depId=010102,sex=男
]
```

你还可以修改 getUserByDepId 的参数，试试传递不同的参数，然后再看看输出的值，看看是否正确地实现了要求的功能。

11.1.3 有何问题

上面的实现看起来很简单，功能也正确，但是蕴涵一个较大的问题。那就是，当一次性访问的数据条数过多，而且每条描述的数据量又很大的话，将会消耗较多的内存。

前面也说了，对于用户表，事实上是有很多字段的，不仅仅是示例的几个，再加上不使用翻页，一次性访问的数据就可能会有很多条。如果一次性需要访问的数据较多，内存开销将会比较大。

但是从客户使用的角度来说，有很大的随机性。客户有可能访问每一条数据，也有可能一条都不访问。也就是说，一次性访问很多条数据，消耗了大量内存，但是很可能是浪费掉了，客户根本就不会去访问那么多数据，对于每条数据，客户只需要看看姓名而已。

> 提示 **那么该怎么实现，才能既把多条用户数据的姓名显示出来，而又能节省内存空间？当然还要实现在客户想要看到更多数据的时候，能正确访问到数据呢！**

11.2 解决方案

11.2.1 使用代理模式来解决问题

用来解决上述问题的一个合理的解决方案就是代理模式。那么什么是代理模式呢？

1. 代理模式的定义

> 为其他对象提供一种代理以控制对这个对象的访问。

2. 应用代理模式来解决问题的思路

仔细分析上面的问题，一次性访问多条数据，这个可能性是很难避免的，是客户的需要。也就是说，要想节省内存，就不能从减少数据条数入手了，那就只能从减少每条数据的数据量上来考虑。

> **提示**　**一个基本的思路如下：由于客户访问多条用户数据的时候，基本上只需要看到用户的姓名，因此可以考虑刚开始从数据库查询返回的用户数据就只有用户编号和用户姓名，当客户想要详细查看某个用户数据的时候，再次根据用户编号到数据库中获取完整的用户数据。这样一来，就可以在满足客户功能的前提下，大大减少对内存的消耗，只是每次需要重新查询一下数据库，算是一个以时间换空间的策略。**

可是该如何来表示这个只有用户编号和姓名的对象呢？它还需要实现在必要的时候访问数据库去重新获取完整的用户数据。

代理模式引入一个 Proxy 对象来解决这个问题。刚开始只有用户编号和姓名的时候，不是一个完整的用户对象，而是一个代理对象。当需要访问完整的用户数据的时候，代理会从数据库中重新获取相应的数据，通常情况下是当客户需要访问除了用户编号和姓名之外的数据的时候，代理才会重新去获取数据。

11.2.2　代理模式的结构和说明

代理模式的结构如图 11.1 所示。

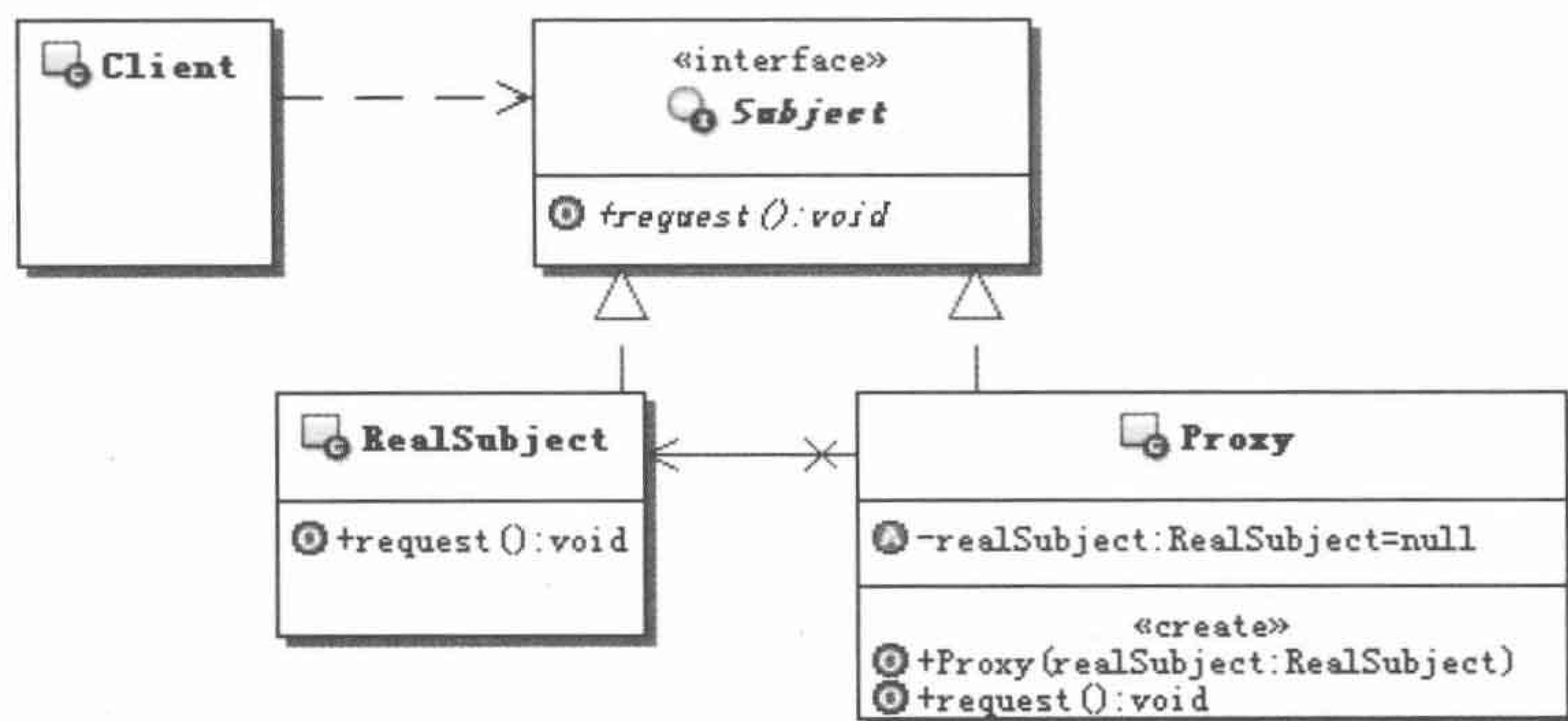


图 11.1　代理模式的结构示意图

- Proxy：代理对象，通常具有如下功能。
  实现与具体的目标对象一样的接口，这样就可以使用代理来代替具体的目标对象。
  保存一个指向具体目标对象的引用，可以在需要的时候调用具体的目标对象。

可以控制对具体目标对象的访问，并可以负责创建和删除它。

- Subject：目标接口，定义代理和具体目标对象的接口，这样就可以在任何使用具体目标对象的地方使用代理对象。
- RealSubject：具体的目标对象，真正实现目标接口要求的功能。

在运行时刻一种可能的代理结构的对象图如图 11.2 所示。

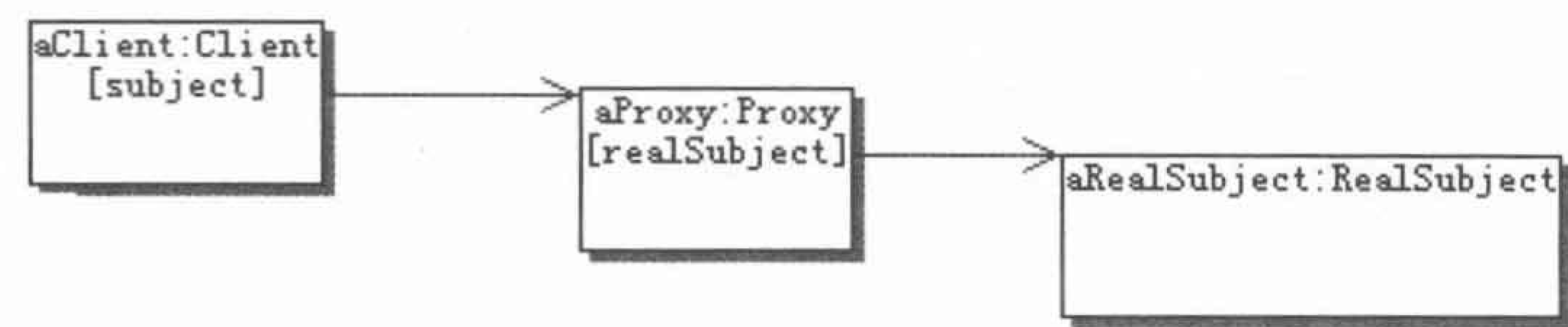


图 11.2　运行时刻一种可能的代理结构的对象图

11.2.3　代理模式示例代码

（1）先看看目标接口的定义。示例代码如下：

```
/**
 * 抽象的目标接口，定义具体的目标对象和代理公用的接口
 */
public interface Subject {
    /**
     * 示意方法：一个抽象的请求方法
     */
    public void request();
}
```

（2）接下来看看具体目标对象的实现示意。示例代码如下：

```
/**
 * 具体的目标对象，是真正被代理的对象
 */
public class RealSubject implements Subject{
    public void request() {
        //执行具体的功能处理
    }
}
```

（3）再来看看代理对象的实现示意。示例代码如下：

```
/**
 * 代理对象
 */
public class Proxy implements Subject{
    /**
     * 持有被代理的具体的目标对象
```

```
     */
    private RealSubject realSubject=null;
    /**
     * 构造方法，传入被代理的具体的目标对象
     * @param realSubject 被代理的具体的目标对象
     */
    public Proxy(RealSubject realSubject){
        this.realSubject = realSubject;
    }
    public void request() {
        //在转调具体的目标对象前，可以执行一些功能处理

        //转调具体的目标对象的方法
        realSubject.request();

        //在转调具体的目标对象后，可以执行一些功能处理
    }
}
```

11.2.4　使用代理模式重写示例

要使用代理模式来重写示例，首先就需要为用户对象定义一个接口，然后实现相应的用户对象的代理。这样在使用用户对象的地方，使用这个代理对象就可以了。

这个代理对象，在起初创建的时候，只需要装载用户编号和姓名这两个基本的数据，然后在客户需要访问除这两个属性外的数据的时候，才再次从数据库中查询并装载数据，从而达到节省内存的目的。因为如果用户不去访问详细的数据，那么这些数据就不需要被装载，对内存的消耗就会减少。

先看看这个时候系统的整体结构，如图 11.3 所示。

此时的 UserManager 类充当了标准代理模式中的 Client 的角色，因为是它在使用代理对象和用户数据对象的接口。

还是看看具体的代码示例，会更清楚。

（1）先看看新定义的用户数据对象的接口，非常简单，就是对用户数据对象属性操作的 getter/setter 方法，因此也没有必要去注释了。示例代码如下：

```
/**
 * 定义用户数据对象的接口
 */
public interface UserModelApi {
    public String getUserId();
    public void setUserId(String userId);
```

```
    public String getName();
    public void setName(String name);
    public String getDepId();
    public void setDepId(String depId);
    public String getSex();
    public void setSex(String sex);
}
```

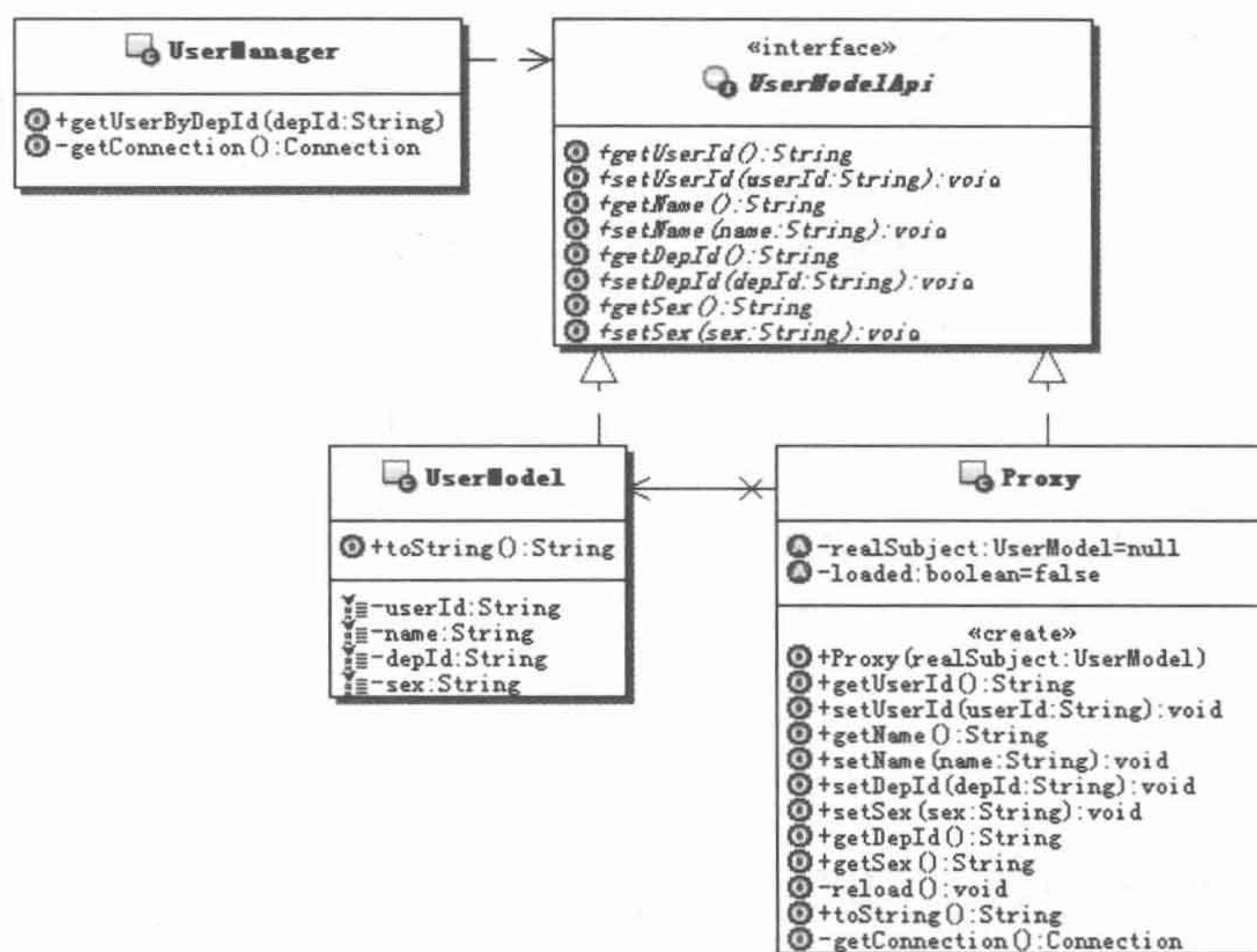


图 11.3　代理模式重写示例的系统结构示意图

（2）定义了接口，需要让 UserModel 来实现它。基本没有什么变化，只是要实现这个新的接口而已，就不再代码示例了。

（3）接下来看看新加入的代理对象的实现。示例代码如下：

```
/**
 * 代理对象，代理用户数据对象
 */
public class Proxy implements UserModelApi{
    /**
     * 持有被代理的具体的目标对象
     */
    private UserModel realSubject=null;
    /**
     * 构造方法，传入被代理的具体的目标对象
     * @param realSubject 被代理的具体的目标对象
     */
    public Proxy(UserModel realSubject){
        this.realSubject = realSubject;
    }
```

```
/**
 * 标示是否已经重新装载过数据了
 */
private boolean loaded = false;
    public String getUserId() {
    return realSubject.getUserId();
}
public void setUserId(String userId) {
    realSubject.setUserId(userId);
}
public String getName() {
    return realSubject.getName();
}
public void setName(String name) {
    realSubject.setName(name);
}
public void setDepId(String depId) {
    realSubject.setDepId(depId);
}
public void setSex(String sex) {
    realSubject.setSex(sex);
}
public String getDepId() {
    //需要判断是否已经装载过了
    if(!this.loaded){
        //从数据库中重新装载
        reload();
        //设置重新装载的标志为true
        this.loaded = true;
    }
    return realSubject.getDepId();
}
public String getSex() {
    if(!this.loaded){
        reload();
        this.loaded = true;
    }
    return realSubject.getSex();
}
/**
```

用户编号和姓名是已经获取到的数据，直接调用具体目标对象的数据就可以了

setter 方法不需要重新查询数据库，直接调用具体目标对象的相应功能就可以了

```
     * 重新查询数据库以获取完整的用户数据
     */
    private void reload(){
        System.out.println("重新查询数据库获取完整的用户数据，userId=="
                                        +realSubject.getUserId());
        Connection conn = null;
        try{
            conn = this.getConnection();
            String sql = "select * from tbl_user where userId=?";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, realSubject.getUserId());
            ResultSet rs = pstmt.executeQuery();
            if(rs.next()){
                //只需要重新获取除了userId和name外的数据
                realSubject.setDepId(rs.getString("depId"));
                realSubject.setSex(rs.getString("sex"));
            }

            rs.close();
            pstmt.close();
        }catch(Exception err){
            err.printStackTrace();
        }finally{
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
    public String toString(){
        return "userId="+getUserId()+",name="+getName()
                +",depId="+getDepId()+",sex="+getSex()+"\n";
    }
    private Connection getConnection() throws Exception {
        Class.forName("你用的数据库对应的JDBC驱动类");
        return DriverManager.getConnection(
                "连接数据库的URL", "用户名", "密码");
    }
}
```

（4）看看此时 UserManager 的变化，大致如下。

- 从数据库查询值的时候，不需要全部获取了，只需要查询用户编号和姓名的数据就可以了。
- 把数据库中获取的值转变成对象的时候，创建的对象不再是 UserModel，而是代理对象，而且设置值的时候，也不是全部都设置，只是设置用户编号和姓名两个属性的值。

示例代码如下：

```
/**
 * 实现示例要求的功能
 */
public class UserManager {
    /**
     * 根据部门编号来获取该部门下的所有人员
     * @param depId 部门编号
     * @return 该部门下的所有人员
     */
    public Collection<UserModelApi> getUserByDepId(
                                    String depId)throws Exception{
        Collection<UserModelApi> col =
                                    new ArrayList<UserModelApi>();
        Connection conn = null;
        try{
            conn = this.getConnection();
            //只需要查询userId和name两个值就可以了
            String sql = "select u.userId,u.name "
                +"from tbl_user u,tbl_dep d "
                +"where u.depId=d.depId and d.depId like ?";

            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, depId+"%");

            ResultSet rs = pstmt.executeQuery();
            while(rs.next()){
                //这里是创建的代理对象，而不是直接创建UserModel的对象
                Proxy proxy = new Proxy(new UserModel());
                //只是设置userId和name两个值就可以了
                proxy.setUserId(rs.getString("userId"));
                proxy.setName(rs.getString("name"));
```

```
                    col.add(proxy);
                }

                rs.close();
                pstmt.close();
            }finally{
                conn.close();
            }
            return col;
        }
        private Connection getConnection() throws Exception {
            Class.forName("你用的数据库对应的JDBC驱动类");
            return DriverManager.getConnection(
                    "连接数据库的URL", "用户名", "密码");
        }
}
```

（5）写个客户端来测试看看，是否能正确实现代理的功能！示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception{
        UserManager userManager = new UserManager();
        Collection<UserModelApi> col =
                                userManager.getUserByDepId("0101");

        //如果只是显示用户名称，则不需要重新查询数据库
        for(UserModelApi umApi : col){
            System.out.println("用户编号：="+umApi.getUserId()
                                    +",用户姓名：="+umApi.getName());
        }
        //如果访问非用户编号和用户姓名外的属性，那就会重新查询数据库
        for(UserModelApi umApi : col){
            System.out.println("用户编号：="+umApi.getUserId()
                                    +",用户姓名：="+umApi.getName()
                                    +",所属部门：="+umApi.getDepId());
        }
    }
}
```

运行结果如下：

```
用户编号：=user0001,用户姓名：=张三1
用户编号：=user0002,用户姓名：=张三2
```

```
用户编号：=user0003,用户姓名：=张三3
重新查询数据库获取完整的用户数据，userId==user0001
用户编号：=user0001,用户姓名：=张三1,所属部门：=010101
重新查询数据库获取完整的用户数据，userId==user0002
用户编号：=user0002,用户姓名：=张三2,所属部门：=010101
重新查询数据库获取完整的用户数据，userId==user0003
用户编号：=user0003,用户姓名：=张三3,所属部门：=010102
```

仔细查看上面的结果数据会发现，如果只是访问用户编号和用户姓名的数据，是不需要重新查询数据库的。只有当访问到这两个数据以外的数据时，才需要重新查询数据库以获得完整的数据。这样一来，如果客户不访问除这两个数据以外的数据，那么就不需要重新查询数据库，也就不需要装载那么多数据，从而节省了内存。

（6）1+N 次查询。

看完上面的示例，可能有些朋友会发现，这种实现方式有一个潜在的问题，就是如果客户对每条用户数据都要求查看详细数据的话，那么总的查询数据库的次数会是 $1+N$ 次之多。

第一次查询，获取到 N 条数据的用户编号和姓名，然后展示给客户看。如果这个时候，客户对每条数据都点击查看详细信息的话，那么每一条数据都需要重新查询数据库，那么最后总的查询数据库的次数就是 $1+N$ 次了。

从上面的分析可以看出，这种做法最合适的场景就是：客户大多数情况下只需要查看用户编号和姓名，而少量的数据需要查看详细数据。这样既节省了内存，又减少了操作数据库的次数。

延伸　**看到这里，可能会有朋友想起，Hibernate 这类 ORM 的框架，在 Lazy Load 的情况下，也存在 1+N次查询的情况，原因就在于，Hibernate 的 Lazy Load 就是使用代理来实现的，具体的实现细节这里就不去讨论了，但是原理是一样的。**

11.3　模式讲解

11.3.1　认识代理模式

1. 代理模式的功能

代理模式是通过创建一个代理对象，用这个代理对象去代表真实的对象，客户端得到这个代理对象后，对客户端并没有什么影响，就跟得到了真实对象一样来使用。

当客户端操作这个代理对象的时候，实际上功能最终还是会由真实的对象来完成，只不过是通过代理操作的，也就是客户端操作代理，代理操作真正的对象。

正是因为有代理对象夹在客户端和被代理的真实对象中间，相当于一个中转，那么在中转的时候就有很多花招可以玩，比如，判断一下权限，如果没有足够的权限那就不

给你中转了，等等。

2. 代理的分类

事实上代理又被分成多种，大致有如下一些。

- 虚代理：根据需要来创建开销很大的对象，该对象只有在需要的时候才会被真正创建。
- 远程代理：用来在不同的地址空间上代表同一个对象，这个不同的地址空间可以是在本机，也可以在其他机器上。在 Java 里面最典型的就是 RMI 技术。
- copy-on-write 代理：在客户端操作的时候，只有对象确实改变了，才会真的拷贝（或克隆）一个目标对象，算是虚代理的一个分支。
- 保护代理：控制对原始对象的访问，如果有需要，可以给不同的用户提供不同的访问权限，以控制他们对原始对象的访问。
- Cache 代理：为那些昂贵操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
- 防火墙代理：保护对象不被恶意用户访问和操作。
- 同步代理：使多个用户能够同时访问目标对象而没有冲突。
- 智能指引：在访问对象时执行一些附加操作，比如，对指向实际对象的引用计数、第一次引用一个持久对象时，将它装入内存等。

在这些代理类型中，最常见的是虚代理、保护代理、远程代理和智能指引这几种。本书主要讨论虚代理和保护代理并给出了示例，这是实际开发中使用频率最高的两种代理。

对于远程代理，没有去讨论，因为在 Java 中，远程代理的典型体现是 RMI 技术。要想把远程代理讲述清楚，就需要把 RMI 讲述清楚，这不在本书的讨论范围之内。

对于智能指引，基本的实现方式和保护代理的实现类似，只是实现的具体功能有所不同，因此也没有具体去讨论和示例。

3. 虚代理的示例

前面的例子就是一个典型的虚代理的实现。

起初每个代理对象只有用户编号和姓名数据，直到需要的时候，才会把整个用户的数据装载到内存中来。

也就是说，要根据需要来装载整个 UserModel 的数据，虽然用户数据对象是前面已经创建好了的，但是只有用户编号和姓名的数据，可以看成是一个“虚”的对象，直到通过代理把所有的数据都设置好，才算是一个完整的用户数据对象。

4. copy-on-write

复制一个大的对象是很消耗资源的，如果这个被复制的对象从上次操作以来，根本就没有被修改过，那么再复制这个对象是没有必要的，只是白白消耗资源而已。于是使用代理来延迟复制的过程，可以等到对象被修改的时候才真正地对它进行复制。

copy-on-write 可以大大降低复制大对象的开销，因此它算是一种优化方式，可以根据需要来复制或者克隆对象。

5. 具体目标和代理的关系

从代理模式的结构图来看，好像是有一个具体目标类就有一个代理类，其实不是这样的。如果代理类能完全通过接口来操作它所代理的目标对象，那么代理对象就不需要知道具体的目标对象，这样就无须为每一个具体目标类都创建一个代理类了。

但是，如果代理类必须要实例化它代理的目标对象，那么代理类就必须知道具体被代理的对象，这种情况下，一个具体目标类通常会有一个代理类。这种情况多出现在虚代理的实现里面。

6. 代理模式调用顺序示意图

代理模式调用顺序如图 11.4 所示。

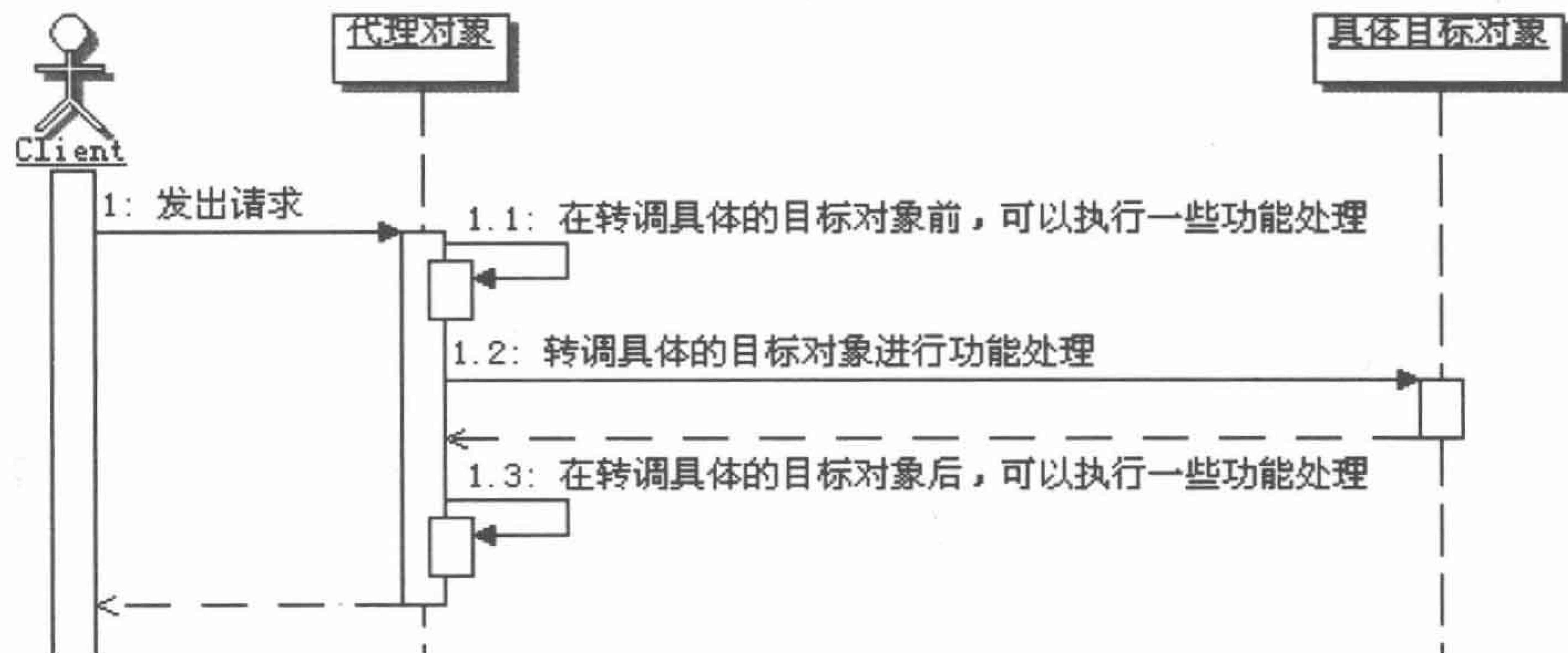


图 11.4　代理模式调用顺序示意图

11.3.2　保护代理

保护代理是一种控制对原始对象访问的代理，多用于对象应该有不同的访问权限的时候。保护代理会检查调用者是否具有请求所必需的访问权限，如果没有相应的权限，那么就不会调用目标对象，从而实现对目标对象的保护。

还是通过一个示例来说明。

1. 示例需求

现在有一个订单系统，要求是：一旦订单被创建，只有订单的创建人才可以修改订单中的数据，其他人则不能修改。

相当于现在如果有了一个订单对象实例，那么就需要控制外部对它的访问，满足条件的可以访问，不满足条件的就不能访问。

2. 示例实现

（1）订单对象的接口定义

要实现这个功能需要，先来定义订单对象的接口。很简单，主要是对订单对象的属性的 getter/setter 方法。示例代码如下：

```
/**
 * 订单对象的接口定义
```

```
 */
public interface OrderApi {
    /**
     * 获取订单订购的产品名称
     * @return 订单订购的产品名称
     */
    public String getProductName();
    /**
     * 设置订单订购的产品名称
     * @param productName 订单订购的产品名称
     * @param user 操作人员
     */
    public void setProductName(String productName,String user);
    /**
     * 获取订单订购的数量
     * @return 订单订购的数量
     */
    public int getOrderNum();
    /**
     * 设置订单订购的数量
     * @param orderNum 订单订购的数量
     * @param user 操作人员
     */
    public void setOrderNum(int orderNum,String user);
    /**
     * 获取创建订单的人员
     * @return 创建订单的人员
     */
    public String getOrderUser();
    /**
     * 设置创建订单的人员
     * @param orderUser 创建订单的人员
     * @param user 操作人员
     */
    public void setOrderUser(String orderUser,String user);
}
```

（2）订单对象

接下来定义订单对象，原本订单对象需要描述的属性很多，为了简单，只描述三个就可以了。示例代码如下：

```
/**
 * 订单对象
 */
public class Order implements OrderApi{
    /**
     * 订单订购的产品名称
     */
    private String productName;
    /**
     * 订单订购的数量
     */
    private int orderNum;
    /**
     * 创建订单的人员
     */
    private String orderUser;

    /**
     * 构造方法，传入构建需要的数据
     * @param productName 订单订购的产品名称
     * @param orderNum 订单订购的数量
     * @param orderUser 创建订单的人员
     */
    public Order(String productName,int orderNum,String orderUser){
        this.productName = productName;
        this.orderNum = orderNum;
        this.orderUser = orderUser;
    }

    public String getProductName() {
        return productName;
    }
    public void setProductName(String productName,String user) {
        this.productName = productName;
    }
    public int getOrderNum() {
        return orderNum;
    }
    public void setOrderNum(int orderNum,String user) {
        this.orderNum = orderNum;
```

```
    }
    public String getOrderUser() {
        return orderUser;
    }
    public void setOrderUser(String orderUser,String user) {
        this.orderUser = orderUser;
    }
}
```

（3）订单对象的代理

创建好了订单对象以后，需要创建对它的代理对象了。既然订单代理就相当于一个订单，那么最自然的方式就是让订单代理跟订单对象实现一样的接口；要控制对订单setter方法的访问，那么就需要在代理的方法里面进行权限判断，有权限则调用订单对象的方法，没有权限则提示错误并返回。示例代码如下：

```
/**
 * 订单的代理对象
 */
public class OrderProxy implements OrderApi{
    /**
     * 持有被代理的具体的目标对象
     */
    private Order order=null;
    /**
     * 构造方法，传入被代理的具体的目标对象
     * @param realSubject 被代理的具体的目标对象
     */
    public OrderProxy(Order realSubject){
        this.order = realSubject;
    }

    public void setProductName(String productName,String user) {
        //控制访问权限，只有创建订单的人员才能够修改
        if(user!=null && user.equals(this.getOrderUser())){
            order.setProductName(productName, user);
        }else{
            System.out.println("对不起"+user
                                +"，您无权修改订单中的产品名称。");
        }
    }
    public void setOrderNum(int orderNum,String user) {
```

```
        //控制访问权限，只有创建订单的人员才能够修改
        if(user!=null && user.equals(this.getOrderUser())){
            order.setOrderNum(orderNum, user);
        }else{
            System.out.println("对不起"+user
                                    +"，您无权修改订单中的订购数量。");
        }
    }
    public void setOrderUser(String orderUser,String user) {
        //控制访问权限，只有创建订单的人员才能够修改
        if(user!=null && user.equals(this.getOrderUser())){
            order.setOrderUser(orderUser, user);
        }else{
            System.out.println("对不起"+user
                                    +"，您无权修改订单中的订购人。");
        }
    }

    public int getOrderNum() {
        return this.order.getOrderNum();
    }
    public String getOrderUser() {
        return this.order.getOrderUser();
    }
    public String getProductName() {
        return this.order.getProductName();
    }
    public String toString(){
        return "productName="+this.getProductName()+",orderNum="
            +this.getOrderNum()+",orderUser="+this.getOrderUser();
    }
}
```

（4）测试代码

一起来看看如何使用刚刚完成的订单代理。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //张三先登录系统创建了一个订单
        OrderApi order = new OrderProxy(
                                new Order("设计模式",100,"张三"));
```

```
        //李四想要来修改，那就会报错
        order.setOrderNum(123, "李四");
        //输出order
        System.out.println("李四修改后订单记录没有变化："+order);
        //张三修改就不会有问题
        order.setOrderNum(123, "张三");
        //再次输出order
        System.out.println("张三修改后，订单记录："+order);
    }
}
```

运行结果如下：

```
对不起李四，您无权修改订单中的订购数量
李四修改后订单记录没有变化：
                    productName=设计模式,orderNum=100,orderUser=张三
张三修改后，订单记录：productName=设计模式,orderNum=123,
                                                orderUser=张三
```

从上面的运行结果可以看出，在通过代理转调目标对象的时候，在代理对象中，对访问的用户进行了权限判断，如果不满足要求，就不会转调目标对象的方法，从而保护了目标对象的方法，只让有权限的人操作。

11.3.3 Java 中的代理

Java 对代理模式提供了内建的支持，在 java.lang.reflect 包下面，提供了一个 Proxy 的类和一个 InvocationHandler 的接口。

通常把前面自己实现的代理模式称为 **Java 的静态代理**。这种实现方式有一个较大的缺点，就是如果 Subject 接口发生变化，那么代理类和具体的目标实现都要变化，不是很灵活。而使用 Java 内建的对代理模式支持的功能来实现则没有这个问题。

通常把使用 Java 内建的对代理模式支持的功能来实现的代理称为 **Java 的动态代理**。动态代理跟静态代理相比，明显的变化是：静态代理实现的时候，在 Subject 接口上定义很多的方法，代理类里面自然也要实现很多方法；而动态代理实现的时候，虽然 Subject 接口上定义了很多方法，但是动态代理类始终只有一个 invoke 方法。这样，当 Subject 接口发生变化的时候，动态代理的接口就不需要跟着变化了。

Java 的动态代理目前只能代理接口，基本的实现是依靠 Java 的反射机制和动态生成 class 的技术，来动态生成被代理的接口的实现对象。具体的内部实现细节这里不去讨论。如果要实现类的代理，可以使用 cglib（一个开源的 Code Generation Library）。

还是来看看示例，那就修改上面保护代理的示例，看看如何使用 Java 的动态代理来实现同样的功能。

（1）订单接口的定义是完全一样的，就不再赘述了。

（2）订单对象的实现，只是添加了一个 toString，以方便测试输出，这里也不去示例了。在前面的示例中，toString 已实现在代理类里面了。

（3）直接看看代理类的实现，大致有如下变化。

- 要实现 InvocationHandler 接口。
- 需要提供一个方法来实现：把具体的目标对象和动态代理绑定起来，并在绑定好过后，返回被代理的目标对象的接口，以利于客户端的操作。
- 需要实现 invoke 方法，在这个方法里面，具体判断当前是在调用什么方法，需要如何处理。

示例代码如下：

```
/**
 * 使用Java中的动态代理
 */
public class DynamicProxy implements InvocationHandler{
    /**
     * 被代理的对象
     */
    private OrderApi order = null;
    /**
     * 获取绑定好代理和具体目标对象后的目标对象的接口
     * @param order 具体的订单对象，相当于具体目标对象
     * @return 绑定好代理和具体目标对象后的目标对象的接口
     */
    public OrderApi getProxyInterface(Order order){
        //设置被代理的对象，好方便invoke里面的操作
        this.order = order;
        //把真正的订单对象和动态代理关联起来
        OrderApi orderApi = (OrderApi) Proxy.newProxyInstance(
                order.getClass().getClassLoader(),
                order.getClass().getInterfaces(),
                this);
        return orderApi;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        //如果是调用setter方法就需要检查权限
        if(method.getName().startsWith("set")){
            //如果不是创建人，那就不能修改
            if(order.getOrderUser()!=null
```

```
                        && order.getOrderUser().equals(args[1])){
                    //可以操作
                    return method.invoke(order, args);
                }else{
                    System.out.println("对不起，"+args[1]
                                        +"，您无权修改本订单中的数据");
                }
            }else{
                //不是调用的setter方法就继续运行
                return method.invoke(order, args);
            }
            return null;
    }
}
```

要想看明白上面的实现，需要熟悉 Java 反射的知识，这里就不再展开了。

（4）看看现在的客户端如何使用这个动态代理。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //张三先登录系统创建了一个订单
        Order order = new Order("设计模式",100,"张三");

        //创建一个动态代理
        DynamicProxy dynamicProxy = new DynamicProxy();
        //然后把订单和动态代理关联起来
        OrderApi orderApi = dynamicProxy.getProxyInterface(order);

        //以下就需要使用被代理过的接口来操作了
        //李四想要来修改，那就会报错
        orderApi.setOrderNum(123, "李四");
        //输出order
        System.out.println("李四修改后订单记录没有变化："+orderApi);
        //张三修改就不会有问题
        orderApi.setOrderNum(123, "张三");
        //再次输出order
        System.out.println("张三修改后，订单记录："+orderApi);
    }
}
```

运行结果如下：

```
对不起，李四，您无权修改本订单中的数据
```

```
李四修改后订单记录没有变化：
                    productName=设计模式,orderNum=100,orderUser=张三
张三修改后，订单记录：productName=设计模式,orderNum=123,
                                                    orderUser=张三
```

运行的结果跟前面完全由自己实现的代理模式是一样的。

事实上，Java 的动态代理还是实现 AOP（面向方面编程）的一个重要手段，AOP 的知识这里暂时不做讲述，大家先了解这一点就可以了。

11.3.4　代理模式的特点

代理模式在客户和被客户访问的对象之间，引入了一定程度的间接性，客户是直接使用代理，让代理来与被访问的对象进行交互。不同的代理类型，这种附加的间接性有不同的用途，也就具有不同的特点。

- 远程代理：隐藏了一个对象存在于不同的地址空间的事实，也即是客户通过远程代理去访问一个对象，根本就不关心这个对象在哪里，也不关心如何通过网络去访问到这个对象。从客户的角度来讲，它只是在使用代理对象而已。
- 虚代理：可以根据需要来创建"大"对象，只有到必须创建对象的时候，虚代理才会创建对象，从而大大加快程序运行速度，并节省资源。通过虚代理可以对系统进行优化。
- 保护代理：可以在访问一个对象的前后，执行很多附加的操作，除了进行权限控制之外，还可以进行很多跟业务相关的处理，而不需要修改被代理的对象。也就是说，可以通过代理来给目标对象增加功能。
- 智能指引：和保护代理类似，也是允许在访问一个对象的前后，执行很多附加的操作，这样一来就可以做很多额外的事情，比如，引用计数等。

11.3.5　思考代理模式

1. 代理模式的本质

> 代理模式的本质：**控制对象访问。**

代理模式通过代理目标对象，把代理对象插入到客户和目标对象之间，从而为客户和目标对象引入一定的间接性。正是这个间接性，给了代理对象很多的活动空间。代理对象可以在调用具体的目标对象前后，附加很多操作，从而实现新的功能或是扩展目标对象的功能。更狠的是，代理对象还可以不去创建和调用目标对象，也就是说，目标对象被完全代理掉了，或是被替换掉了。

从实现上看，代理模式主要是使用对象的组合和委托，尤其是在静态代理的实现里面，会看得更清楚。但是也可以采用对象继承的方式来实现代理，这种实现方式在某些

情况下，比使用对象组合还要来得简单。

举个例子来说明一下。改造“11.3.2 保护代理”中的例子来说明。

（1）首先去掉 OrderApi，现在改成继承的方式实现代理，不再需要公共的接口了。

（2）Order 对象变化不大，只是去掉实现的 OrderApi 接口就可以了。示例代码如下：

```
public class Order implements OrderApi{
    //其他的代码没有任何变化，就不再赘述了
}
```

（3）再看看代理的实现，变化较多，大致有如下的变化。

- 不再实现 OrderApi，而改成继承 Order。
- 不需要再持有目标对象了，因为这个时候父类就是被代理的对象。
- 原来的构造方法去掉，重新实现一个传入父类需要的数据的构造方法。
- 原来转调目标对象的方法，现在变成调用父类的方法了，用 super 关键字。
- 除了几个被保护代理的 setter 方法外，不再需要 getter 方法了。

示例代码如下：

```
/**
 * 订单的代理对象
 */
public class OrderProxy extends Order{
    public OrderProxy(String productName
                         ,int orderNum,String orderUser){
        super(productName,orderNum,orderUser);
    }
    public void setProductName(String productName,String user) {
        //控制访问权限，只有创建订单的人员才能够修改
        if(user!=null && user.equals(this.getOrderUser())){
            super.setProductName(productName, user);
        }else{
            System.out.println("对不起"+user
                         +"，您无权修改订单中的产品名称。");
        }
    }
    public void setOrderNum(int orderNum,String user) {
        //控制访问权限，只有创建订单的人员才能够修改
        if(user!=null && user.equals(this.getOrderUser())){
            super.setOrderNum(orderNum, user);
        }else{
            System.out.println("对不起"+user
                         +"，您无权修改订单中的订购数量。");
```

```
        }
    }
    public void setOrderUser(String orderUser,String user) {
        //控制访问权限，只有创建订单的人员才能够修改
        if(user!=null && user.equals(this.getOrderUser())){
            super.setOrderUser(orderUser, user);
        }else{
            System.out.println("对不起"+user
                                +"，您无权修改订单中的订购人。");
        }
    }
    public String toString(){
        return "productName="+this.getProductName()+",orderNum="
            +this.getOrderNum()+",orderUser="+this.getOrderUser();
    }
}
```

（4）客户端的变化不大，主要是不再直接面向 OrderApi 接口，而是使用 Order 对象了。另外创建代理的构造方法也发生了变化。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //张三先登录系统创建了一个订单
        Order order = new OrderProxy("设计模式",100,"张三");

        //李四想要来修改，那就会报错
        order.setOrderNum(123, "李四");
        //输出order
        System.out.println("李四修改后订单记录没有变化："+order);

        //张三修改就不会有问题
        order.setOrderNum(123, "张三");
        //再次输出order
        System.out.println("张三修改后，订单记录："+order);
    }
}
```

运行一下，测试看看，体会一下这种实现方式。

2. 何时选用代理模式

建议在如下情况中选用代理模式。

- 需要为一个对象在不同的地址空间提供局部代表的时候，可以使用远程代理。
- 需要按照需要创建开销很大的对象的时候，可以使用虚代理。

- 需要控制对原始对象的访问的时候，可以使用保护代理。
- 需要在访问对象执行一些附加操作的时候，可以使用智能指引代理。

11.3.6 相关模式

- 代理模式和适配器模式

  这两个模式有一定的相似性，但也有差异。

  这两个模式有相似性，它们都为另一个对象提供间接性的访问，而且都是从自身以外的一个接口向这个对象转发请求。

  但是从功能上，两个模式是不一样的。适配器模式主要用来解决接口之间不匹配的问题，它通常是为所适配的对象提供一个不同的接口；而代理模式会实现和目标对象相同的接口。
- 代理模式和装饰模式

  这两个模式从实现上相似，但是功能上是不同的。

  装饰模式的实现和保护代理的实现上是类似的，都是在转调其他对象的前后执行一定的功能。但是它们的目的和功能都是不同的。

  装饰模式的目的是为了让你不生成子类就可以给对象添加职责，也就是为了动态地增加功能；而代理模式的主要目的是控制对对象的访问。

第 12 章　观察者模式（Observer）

12.1 场景问题

12.1.1 订阅报纸的过程

来考虑实际生活中订阅报纸的过程，这里简单总结了一下订阅报纸的基本流程，如下：

（1）首先按照自己的需要选择合适的报纸，具体的报刊杂志目录可以从邮局获取。

（2）选择好后，就到邮局去填写订阅单，同时交纳所需的费用。

至此，就完成了报纸的订阅过程，接下来就是耐心等候，报社会按照出报时间推出报纸，然后报纸会被送到每个订阅人的手里。

画个图来描述上述过程，如图 12.1 所示。

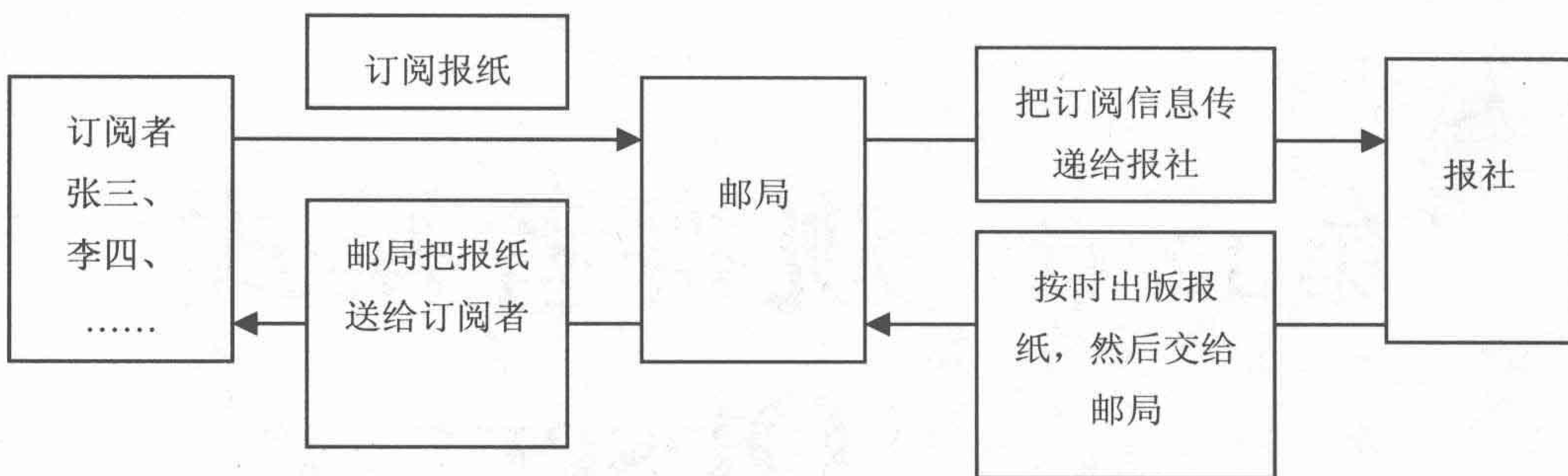


图 12.1　订阅报纸的过程示意图

虽然看起来订阅者是直接跟邮局在打交道，但实际上，订阅者的订阅数据是会被邮局传递到报社的，当报社出版了报纸，报社会按照订阅信息把报纸交给邮局，然后由邮局来代为发送到订阅者的手中。所以在整个过程中，邮局只不过起到一个中转的作用。为了简单，我们去掉邮局，让订阅者直接和报社交互，如图 12.2 所示。

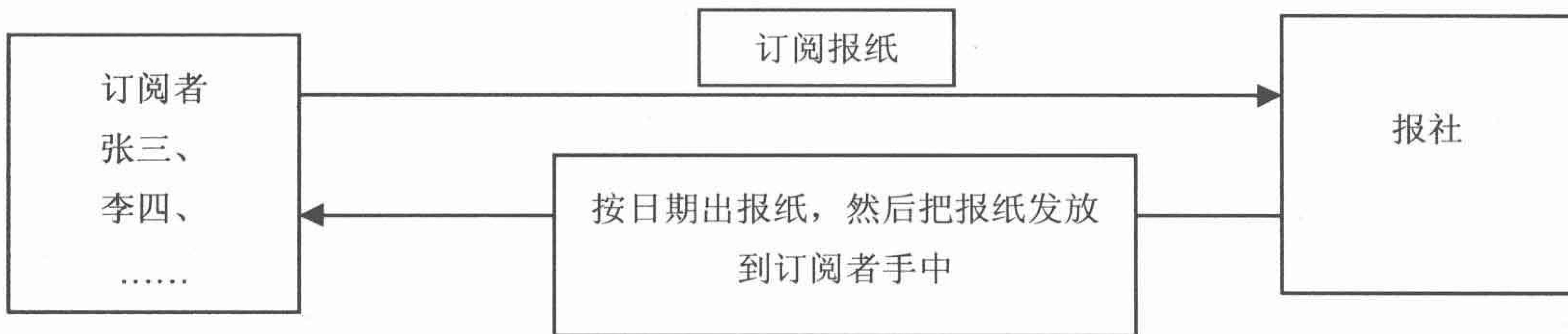


图 12.2　简化的订阅报纸过程示意图

12.1.2 订阅报纸的问题

在上述过程中，订阅者在完成订阅后，最关心的问题就是何时能收到新出的报纸。幸好在现实生活中，报纸都是定期出版，这样发放到订阅者手中也基本上有一个大致的时间范围，差不多到时间了，订阅者就会看看邮箱，查收新的报纸。

要是报纸出版的时间不固定呢？

那订阅者就麻烦了，如果订阅者想要第一时间阅读到新报纸，恐怕只能天天守着邮箱了，这未免也太痛苦了吧。

继续引申一下，用类来描述上述的过程，描述如下。

订阅者类向出版者类订阅报纸，很明显不会只有一个订阅者订阅报纸，订阅者类可以有很多；当出版者类出版新报纸的时候，多个订阅者类如何知道呢？还有订阅者类如何得到新报纸的内容呢？

把上面的问题对比描述一下：

具体描述	对应的抽象描述
当报社有新报纸出版的时候	当出版者类的状态发生改变的时候
多个订阅报纸的人员	多个订阅者类
如何知道?	如何能得到通知?
订阅报纸的人员需要得到新报纸的内容，要看这些新内容	订阅者类会相应进行什么样的处理或改变

进一步抽象描述这个问题：当一个对象的状态发生改变的时候，如何让依赖于它的所有对象得到通知，并进行相应的处理呢？

该如何解决这样的问题？

12.2 解决方案

12.2.1 使用观察者模式来解决问题

用来解决上述问题的一个合理的解决方案就是观察者模式。那么什么是观察者模式呢？

1. 观察者模式的定义

> 定义对象间的一种一对多的依赖关系。当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

2. 应用观察者模式来解决的思路

在前面描述的订阅报纸的例子里面，对于报社来说，在一开始，它并不清楚究竟有多少个订阅者会来订阅报纸，因此，报社需要维护一个订阅者的列表，这样，当报社出版报纸的时候，才能够把报纸发放到所有的订阅者手中。对于订阅者来说，订阅者也就是看报的读者，多个订阅者会订阅同一份报纸。

这就出现了一个典型的一对多的对象关系，一个报纸对象，会有多个订阅者对象来订阅；当报纸出版的时候，也就是报纸对象改变的时候，需要通知所有的订阅者对象。那么怎么来建立并维护这样的关系呢？

观察者模式可以处理这种问题。观察者模式把这多个订阅者称为观察者：Observer，多个观察者观察的对象被称为目标：Subject。

一个目标可以有任意多个观察者对象，一旦目标的状态发生了改变，所有注册的观察者都会得到通知，然后各个观察者会对通知作出相应的响应，执行相应的业务功能处理，并使自己的状态和目标对象的状态保持一致。

12.2.2 观察者模式的结构和说明

观察者模式的结构如图 12.3 所示。

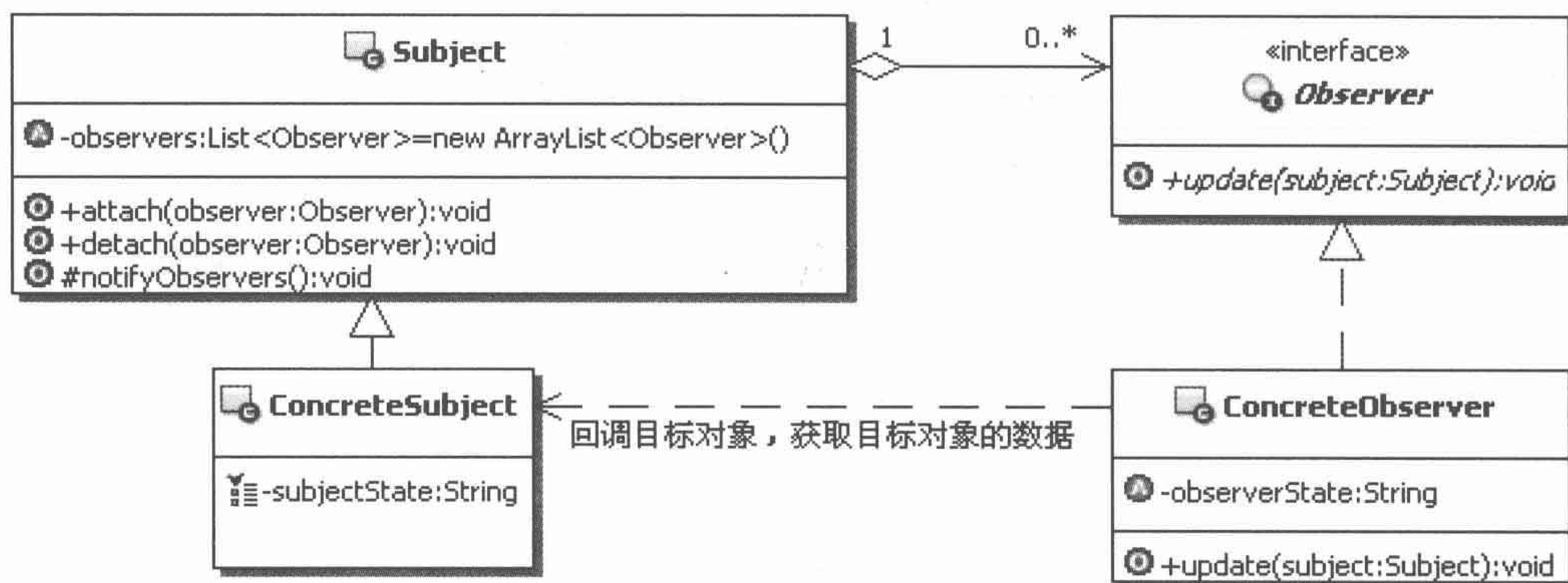


图 12.3 观察者模式的结构示意图

- Subject：目标对象，通常具有如下功能。
  - 一个目标可以被多个观察者观察。
  - 目标提供对观察者注册和退订的维护。
  - 当目标的状态发生变化时，目标负责通知所有注册的、有效的观察者。
- Observer：定义观察者的接口，提供目标通知时对应的更新方法，这个更新方法进行相应的业务处理，可以在这个方法里面回调目标对象，以获取目标对象的数据。
- ConcreteSubject：具体的目标实现对象，用来维护目标状态，当目标对象的状态发生改变时，通知所有注册的、有效的观察者，让观察者执行相应的处理。
- ConcreteObserver：观察者的具体实现对象，用来接收目标的通知，并进行相应的后续处理，比如更新自身的状态以保持和目标的相应状态一致。

12.2.3　观察者模式示例代码

（1）先来看看目标对象的定义。示例代码如下：

```
/**
 * 目标对象，它知道观察它的观察者，并提供注册和删除观察者的接口
 */
public class Subject {
    /**
     * 用来保存注册的观察者对象
     */
    private List<Observer> observers = new ArrayList<Observer>();
    /**
     * 注册观察者对象
     * @param observer 观察者对象
     */
    public void attach(Observer observer) {
        observers.add(observer);
    }
    /**
     * 删除观察者对象
     * @param observer 观察者对象
     */
    public void detach(Observer observer) {
        observers.remove(observer);
    }
    /**
     * 通知所有注册的观察者对象
     */
    protected void notifyObservers() {
        for(Observer observer : observers){
            observer.update(this);
        }
    }
}
```

（2）接下来看看具体的目标对象。示例代码如下：

```
/**
 * 具体的目标对象，负责把有关状态存入到相应的观察者对象
 * 并在自己状态发生改变时，通知各个观察者
 */
```

```
public class ConcreteSubject extends Subject {
    /**
     * 示意，目标对象的状态
     */
    private String subjectState;
    public String getSubjectState() {
        return subjectState;
    }
    public void setSubjectState(String subjectState) {
        this.subjectState = subjectState;
        //状态发生了改变，通知各个观察者
        this.notifyObservers();
    }
}
```

（3）再来看看观察者的接口定义。示例代码如下：

```
/**
 * 观察者接口，定义一个更新的接口给那些在目标发生改变的时候被通知的对象
 */
public interface Observer {
    /**
     * 更新的接口
     * @param subject 传入目标对象，方便获取相应的目标对象的状态
     */
    public void update(Subject subject);
}
```

（4）最后来看看观察者的具体实现示意。示例代码如下：

```
/**
 * 具体观察者对象，实现更新的方法，使自身的状态和目标的状态保持一致
 */
public class ConcreteObserver implements Observer {
    /**
     * 示意，观察者的状态
     */
    private String observerState;

    public void update(Subject subject) {
        // 具体的更新实现
        //这里可能需要更新观察者的状态，使其与目标的状态保持一致
        observerState = ((ConcreteSubject)subject)
```

```
.getSubjectState();
    }
}
```

12.2.4　使用观察者模式实现示例

要使用观察者模式来实现示例，那就按照前面讲述的实现思路，把报纸对象当作目标，订阅者当做观察者，就可以实现出来了。

使用观察者模式来实现示例的结构如图 12.4 所示。

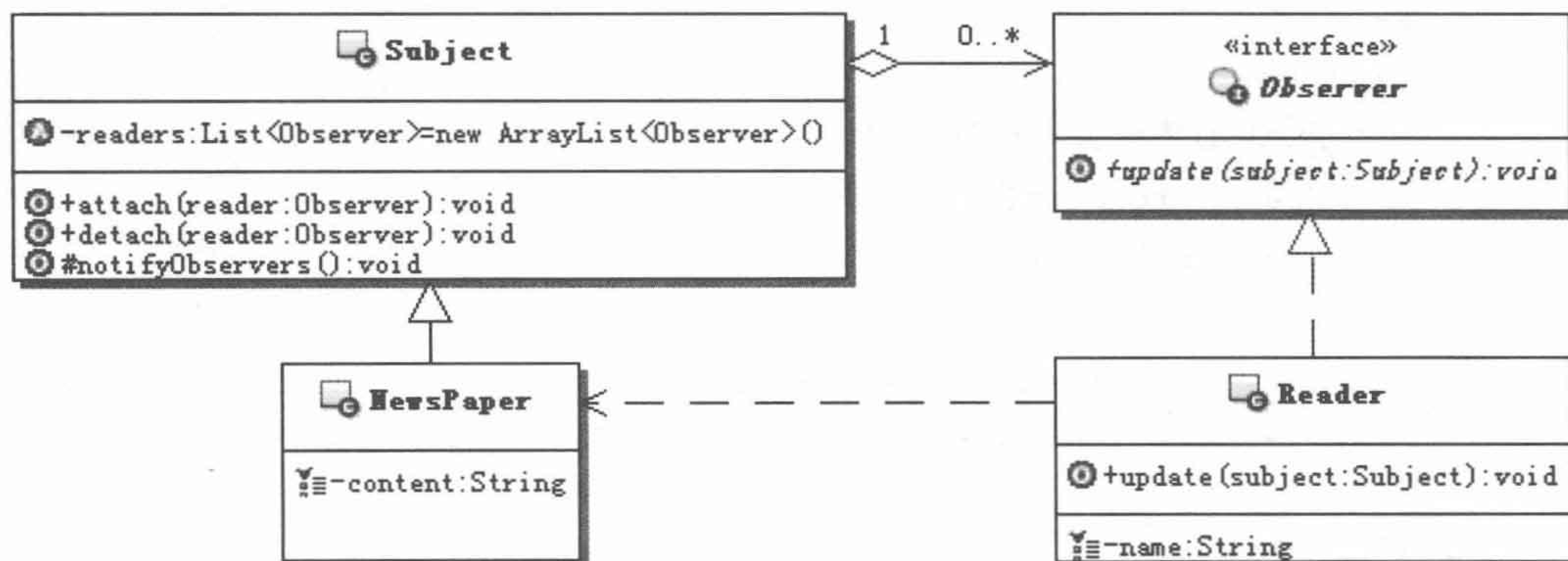


图 12.4　使用观察者模式来实现示例的结构示意图

还是来看看具体的代码实现。

1. 被观察的目标

在前面描述的订阅报纸的例子里面，多个订阅者都是在观察同一个报社对象，这个报社对象就是被观察的目标。这个目标的接口应该有些什么方法呢？还是从实际入手去想，看看报社都有些什么功能。报社有以下最基本的功能。

- 注册订阅者，也就是说很多个人来订报纸，报社肯定要有相应的记录才行。
- 出版报纸，这个是报社的主要工作。
- 发行报纸，也就是要把出版的报纸发送到订阅者手中。
- 退订报纸，当订阅者不想继续订阅了，可以取消订阅。

上面这些功能是报社最基本的功能，当然，报社还有很多别的功能，为了简单，这里就不再去描述了。因此报社这个目标的接口也应该实现上述功能，把它们定义在目标接口里面。示例代码如下：

```
/**
 * 目标对象，作为被观察者
 */
public class Subject {
    /**
     * 用来保存注册的观察者对象，也就是报纸的订阅者
     */
```

```
    private List<Observer> readers = new ArrayList<Observer>();

    /**
     * 报纸的读者需要向报社订阅，先要注册
     * @param reader 报纸的读者
     * @return 是否注册成功
     */
    public void attach(Observer reader) {
        readers.add(reader);
    }
    /**
     * 报纸的读者可以取消订阅
     * @param reader 报纸的读者
     * @return 是否取消成功
     */
    public void detach(Observer reader) {
        readers.remove(reader);
    }
    /**
     * 当每期报纸印刷出来后，就要迅速主动地被送到读者的手中
     * 相当于通知读者，让他们知道
     */
    protected void notifyObservers() {
        for(Observer reader : readers){
            reader.update(this);
        }
    }
}
```

细心的朋友可能会发现，这个对象并没有定义出版报纸的功能，这是为了让这个对象更加通用，这个功能还是有的，放到具体的报纸类中了。下面就来具体地看看具体的报纸类的实现。

为了演示的方便，在这个实现类中增添一个属性，用它来保存报纸的内容，然后增添一个方法来修改这个属性，修改这个属性就相当于出版了新的报纸，并且同时通知所有的订阅者。示例代码如下：

```
/**
 * 报纸对象，具体的目标实现
 */
public class NewsPaper extends Subject{
    /**
```

```
     * 报纸的具体内容
     */
    private String content;
    /**
     * 获取报纸的具体内容
     * @return 报纸的具体内容
     */
    public String getContent() {
        return content;
    }

    /**
     * 示意，设置报纸的具体内容，相当于要出版报纸了
     * @param content 报纸的具体内容
     */
    public void setContent(String content) {
        this.content = content;
        //内容有了，说明又出报纸了，那就通知所有的读者
        notifyObservers();
    }
}
```

2. 观察者

目标定义好以后，接下来把观察者抽象出来，看看他应该具有什么功能。分析前面的描述，发现观察者只要去邮局注册了以后，等着接收报纸就可以了，没有其他的功能了。那么就把这个接收报纸的功能抽象成为更新的方法，从而定义出观察者接口来。

示例代码如下：

```
/**
 * 观察者，比如报纸的读者
 */
public interface Observer {
    /**
     * 被通知的方法
     * @param subject 具体的目标对象，可以获取报纸的内容
     */
    public void update(Subject subject);
}
```

定义好观察者的接口以后，该来想想如何实现了。具体的观察者需要实现：在收到被通知的内容后，自身如何进行相应处理的功能。为了演示的简单，收到报纸内容以后，简单地输出一下，表示收到了就可以了。

定义一个简单的观察者实现。示例代码如下：

```
/**
 * 真正的读者，为了简单就描述一下姓名
 */
public class Reader implements Observer{
    /**
     * 读者的姓名
     */
    private String name;

    public void update(Subject subject) {
        //这是采用拉的方式
        System.out.println(name+"收到报纸了，阅读它。内容是==="
                              +((NewsPaper)subject).getContent());
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

最简单的处理，输出了接收到的内容

3. 使用观察者模式

前面定义好了观察者和观察的目标，那么如何使用它们呢？

那就写个客户端，在客户端里面，先创建好一个报纸，作为被观察的目标，然后多创建几个读者作为观察者，当然需要把这些观察者都注册到目标里面去，接下来就可以出版报纸了。具体的示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建一个报纸，作为被观察者
        NewsPaper subject = new NewsPaper();
        //创建阅读者，也就是观察者
        Reader reader1 = new Reader();
        reader1.setName("张三");

        Reader reader2 = new Reader();
        reader2.setName("李四");
```

```
        Reader reader3 = new Reader();
        reader3.setName("王五");

        //注册阅读者
        subject.attach(reader1);
        subject.attach(reader2);
        subject.attach(reader3);

        //要出报纸啦
        subject.setContent("本期内容是观察者模式");
    }
}
```

测试一下看看，输出结果如下：

```
张三收到报纸了，阅读它。内容是===本期内容是观察者模式
李四收到报纸了，阅读它。内容是===本期内容是观察者模式
王五收到报纸了，阅读它。内容是===本期内容是观察者模式
```

你还可以通过改变注册的观察者，或者是注册了又退订，来看看输出的结果。会发现没有注册或者退订的观察者是收不到报纸的。

如同前面的示例，读者和报社是一种典型的一对多的关系，一个报社有多个读者，当报社的状态发生改变，也就是出版新报纸的时候，所有注册的读者都会得到通知，然后读者会拿到报纸，读者会去阅读报纸并进行后续的操作。

12.3 模式讲解

12.3.1 认识观察者模式

1. 目标和观察者之间的关系

按照模式的定义，目标和观察者之间是典型的一对多的关系。

但是要注意，如果观察者只有一个，也是可以的，这样就变相实现了目标和观察者之间一对一的关系，这也使得在处理一个对象的状态变化会影响到另一个对象的时候，也可以考虑使用观察者模式。

同样地，一个观察者也可以观察多个目标，如果观察者为多个目标定义的通知更新方法都是 update 方法的话，这会带来麻烦，因为需要接收多个目标的通知，如果是一个 update 的方法，那就需要在方法内部区分，到底这个更新的通知来自于哪一个目标，不同的目标有不同的后续操作。

一般情况下，观察者应该为不同的观察者目标定义不同的回调方法，这样实现最简单，不需要在 update 方法内部进行区分。

2. 单向依赖

在观察者模式中，观察者和目标是单向依赖的，只有观察者依赖于目标，而目标是不会依赖于观察者的。

它们之间联系的主动权掌握在目标手中，只有目标知道什么时候需要通知观察者。在整个过程中，观察者始终是被动的，被动地等待目标的通知，等待目标传值给它。

对目标而言，所有的观察者都是一样的，目标会一视同仁地对待。当然也可以通过在目标中进行控制，实现有区别地对待观察者，比如某些状态变化，只需要通知部分观察者，但那是属于稍微变形的用法了，不属于标准的、原始的观察者模式了。

3. 基本的实现说明

- 具体的目标实现对象要能维护观察者的注册信息，最简单的实现方案就如同前面的例子那样，采用一个集合来保存观察者的注册信息。
- 具体的目标实现对象需要维护引起通知的状态，一般情况下是目标自身的状态。变形使用的情况下，也可以是别的对象的状态。
- 具体的观察者实现对象需要能接收目标的通知，能够接收目标传递的数据，或者是能够主动去获取目标的数据，并进行后续处理。
- 如果是一个观察者观察多个目标，那么在观察者的更新方法里面，需要去判断是来自哪一个目标的通知。一种简单的解决方案就是扩展 update 方法，比如在方法里面多传递一个参数进行区分等；还有一种更简单的方法，那就是干脆定义不同的回调方法。

4. 命名建议

- 观察者模式又被称为发布——订阅模式。
- 目标接口的定义，建议在名称后面跟 Subject。
- 观察者接口的定义，建议在名称后面跟 Observer。
- 观察者接口的更新方法，建议名称为 update，当然方法的参数可以根据需要定义，参数个数不限、参数类型不限。

5. 触发通知的时机

> 提示
> **在实现观察者模式的时候，一定要注意触发通知的时机。一般情况下，是在完成了状态维护后触发，因为通知会传递数据，不能够先通知后改数据，这很容易出问题，会导致观察者和目标对象的状态不一致。**

比如，目标一发出通知，就有观察者来取值，结果目标还没有更新数据，这就明显地造成了错误。如下示例就是有问题的。示例代码如下：

```
public void setContent(String content) {
//目标先发出通知了，然后才修改自己的数据，这会造成问题
    notifyAllReader();
    this.content = content;
}
```

6. 相互观察

在某些应用中，可能会出现目标和观察者相互观察的情况。什么意思呢？比如有两套观察者模式的应用，其中一套观察者模式的实现是 A 对象、B 对象观察 C 对象；在另一套观察者模式的实现里面，实现的是 B 对象、C 对象观察 A 对象，那么 A 对象和 C 对象就是在相互观察。

换句话说，A 对象的状态变化会引起 C 对象的联动操作，反过来，C 对象的状态变化也会引起 A 对象的联动操作。对于出现这种状况，要特别小心处理，因为可能会出现死循环的情况。

7. 观察者模式的调用顺序示意图

在使用观察者模式时，会很明显地分成两个阶段，第一个阶段是准备阶段，也就是维护目标和观察者关系的阶段，这个阶段的调用顺序如图 12.5 所示。

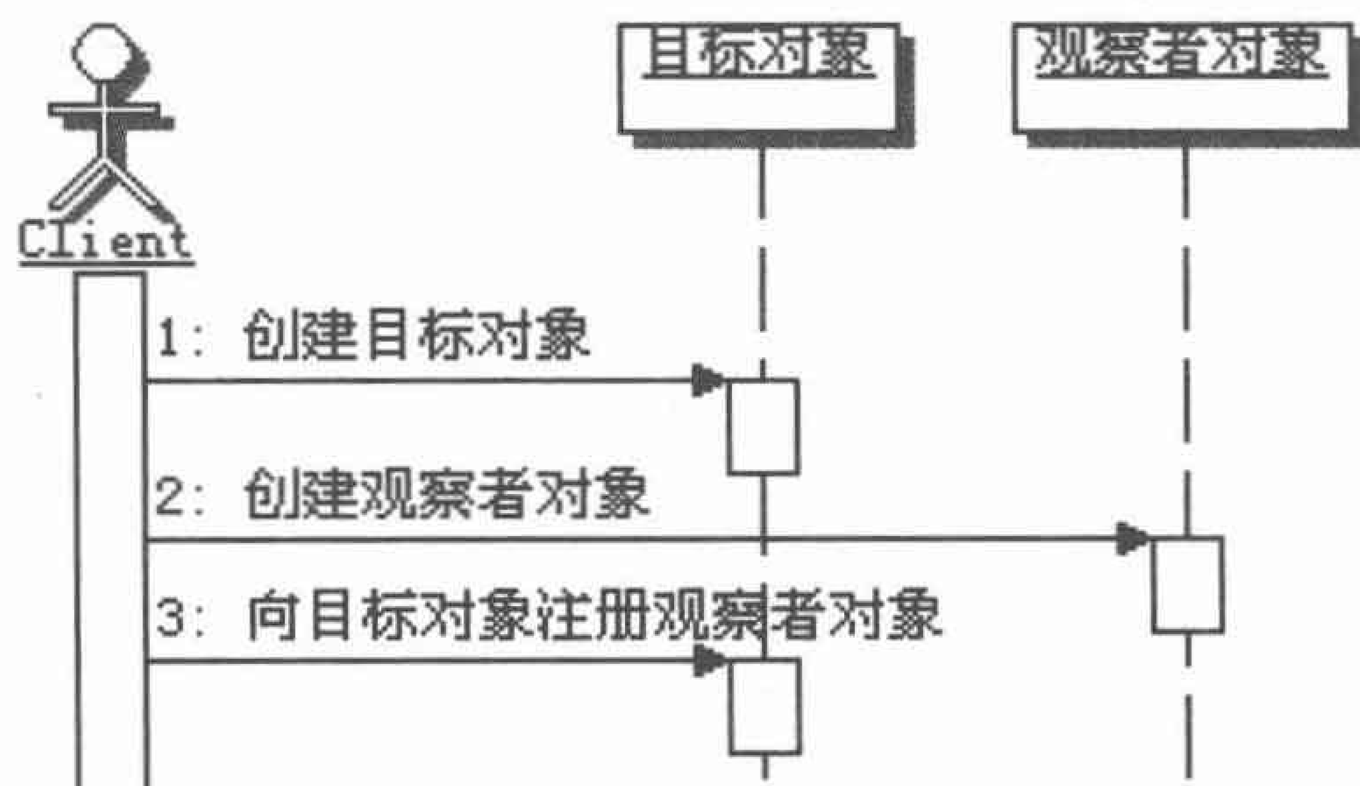


图 12.5 观察者模式准备阶段示意图

接下来就是实际的运行阶段了，这个阶段的调用顺序如图 12.6 所示。

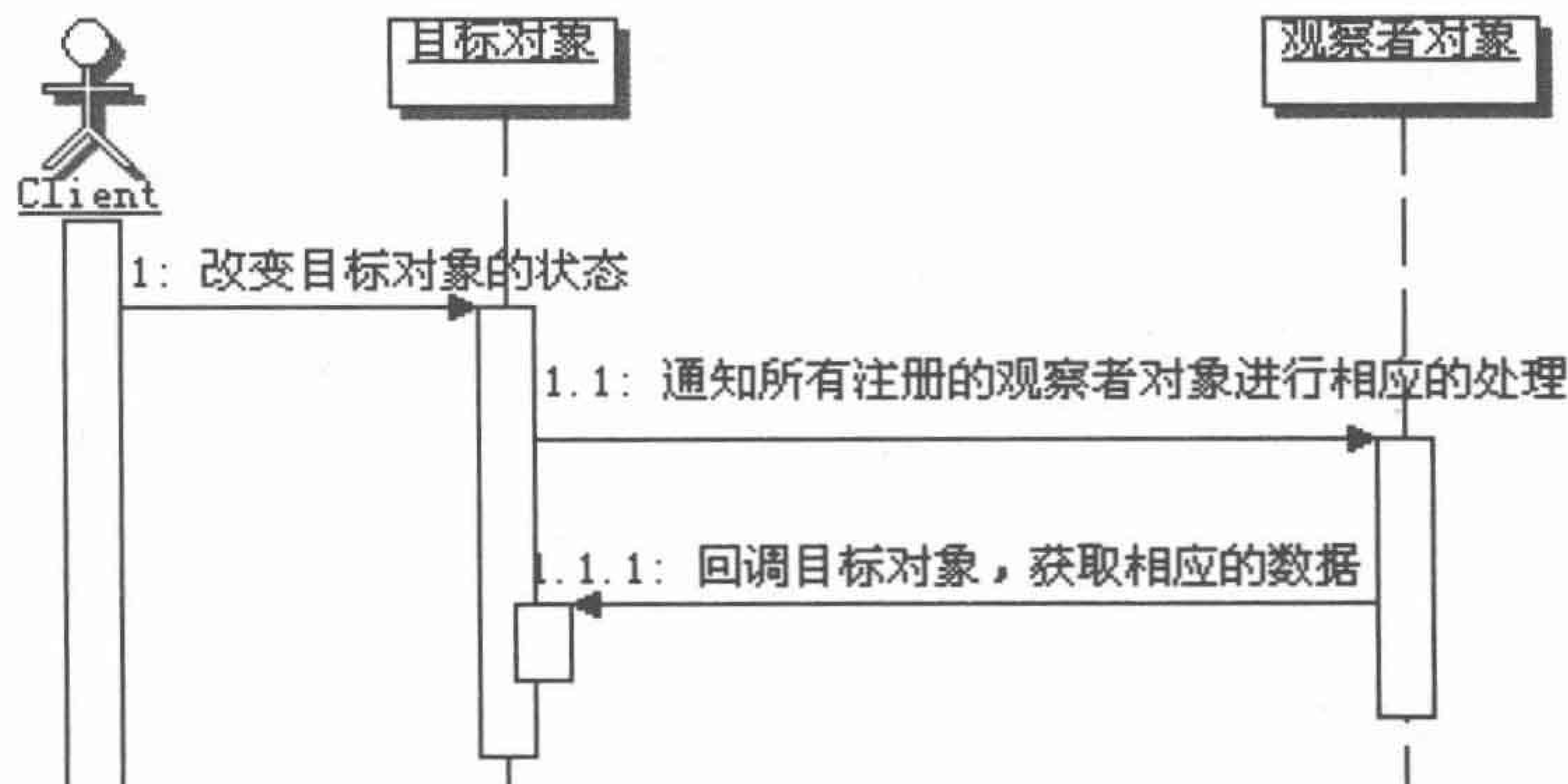


图 12.6 观察者模式运行阶段示意图

8. 通知的顺序

从理论上来说，当目标对象的状态变化后通知所有观察者的时候，顺序是不确定的，因此观察者实现的功能，绝对不要依赖于通知的顺序。也就是说，多个观察者之间的功能是平行的，相互不应该有先后的依赖关系。

12.3.2 推模型和拉模型

在观察者模式的实现中，又分为推模型和拉模型两种方式。什么意思呢？

- 推模型

  目标对象主动向观察者推送目标的详细信息，不管观察者是否需要，推送的信息通常是目标对象的全部或部分数据，相当于是在广播通信。

- 拉模型

  目标对象在通知观察者的时候，只传递少量信息。如果观察者需要更具体的信息，由观察者主动到目标对象中获取，相当于是观察者从目标对象中拉数据。一般这种模型的实现中，会把目标对象自身通过 update 方法传递给观察者，这样在观察者需要获取数据的时候，就可以通过这个引用来获取了。

根据上面的描述，发现前面的例子就是典型的拉模型，那么推模型如何实现呢？还是来看个示例吧，这样会比较清楚。

（1）推模型的观察者接口

根据前面的讲述，推模型通常都是把需要传递的数据直接推送给观察者对象，所以观察者接口中的 update 方法的参数需要发生变化。示例代码如下：

```
/**
 * 观察者，比如报纸的读者
 */
public interface Observer {
    /**
     * 被通知的方法，直接把报纸的内容推送过来
     * @param content 报纸的内容
     */
    public void update(String content);
}
```

（2）推模型的观察者的具体实现

以前需要到目标对象中获取自己需要的数据，现在是直接接收传入的数据，这就是改变的地方。示例代码如下：

```
public class Reader implements Observer{
    /**
     * 读者的姓名
     */
    private String name;

    public void update(String content) {
        //这是采用推的方式
        System.out.println(name+"收到报纸了，阅读它。内容是==="
                      +content);
    }
    public String getName() {
        return name;
```

```
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

（3）推模型的目标对象

跟拉模型的目标实现相比，有一些变化。

- 通知所有观察者的方法，以前是没有参数的，现在需要传入需要主动推送的数据。
- 在循环通知观察者的时候，也就是循环调用观察者的 update 方法的时候，传入的参数不同了。

示例代码如下：

```
/**
 * 目标对象，作为被观察者，使用推模型
 */
public class Subject {
    /**
     * 用来保存注册的观察者对象，也就是报纸的订阅者
     */
    private List<Observer> readers = new ArrayList<Observer>();
    /**
     * 报纸的读者需要向报社订阅，先要注册
     * @param reader 报纸的读者
     * @return 是否注册成功
     */
    public void attach(Observer reader) {
        readers.add(reader);
    }
    /**
     * 报纸的读者可以取消订阅
     * @param reader 报纸的读者
     * @return 是否取消成功
     */
public void detach(Observer reader) {
        readers.remove(reader);
    }
    /**
     * 当每期报纸印刷出来后，就要迅速地、主动地被送到读者的手中
     * 相当于通知读者，让他们知道
     * @param content 要主动推送的内容
```

```
    */
    protected void notifyObservers(String content) {
        for(Observer reader : readers){
            reader.update(content);
        }
    }
}
```

（4）推模型的目标具体实现

跟拉模型相比，有一点变化，就是在调用通知观察者的方法的时候，需要传入参数了，拉模型的实现中是不需要的。示例代码如下：

```
public class NewsPaper extends Subject{
    private String content;
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
        //内容有了，说明又出报纸了，那就通知所有的读者
        notifyObservers(content);
    }
}
```

（5）推模型的客户端使用

跟拉模型一样，没有变化。

好了，到此就简单地实现了推模型的观察者模式，测试一下看看效果，是不是和前面的拉模型一样呢？如果是一样的，那就可以了。

（6）关于两种模型的比较

两种实现模型，在开发的时候，究竟应该使用哪一种，还应该具体问题具体分析。这里，只是把两种模型进行一个简单的比较。

- 推模型是假定目标对象知道观察者需要的数据；而拉模型是目标对象不知道观察者具体需要什么数据，没有办法的情况下，干脆把自身传给观察者，让观察者自己去按需取值。
- 推模型可能会使得观察者对象难以复用，因为观察者定义的 update 方法是按需而定义的，可能无法兼顾没有考虑到的使用情况。这就意味着出现新情况的时候，就可能需要提供新的 update 方法，或者是干脆重新实现观察者。

而拉模型就不会造成这样的情况，因为拉模型下，update 方法的参数是目标对象本身，这基本上是目标对象能传递的最大数据集合了，基本上可以适应各种情况的需要。

12.3.3　Java 中的观察者模式

估计有些朋友在看前面的内容的时候，心里就嘀咕了，Java 中不是已经有了观察者模式的部分实现吗？为何还要全部自己从头做呢？

主要是为了让大家更好地理解观察者模式本身，而不用受 Java 语言实现的限制。

好了，下面就来看看如何利用 Java 中已有的功能来实现观察者模式。在 java.util 包里面有一个类 Observable，它实现了大部分我们需要的目标的功能；还有一个接口 Observer，其中定义了 update 的方法，就是观察者的接口。

因此，利用 Java 中已有的功能来实现观察者模式非常简单，同前面完全由自己来实现观察者模式相比有以下改变。

- 不需要再定义观察者和目标的接口了，JDK 帮忙定义了。
- 具体的目标实现里面不需要再维护观察者的注册信息了，这个在 Java 中的 Observable 类里面，已经帮忙实现好了。
- 触发通知的方式有一点变化，要先调用 setChanged 方法，这个是 Java 为了帮助实现更精确的触发控制而提供的功能。
- 具体观察者的实现里面，update 方法其实能同时支持推模型和拉模型，这个是 Java 在定义的时候，就已经考虑进去了。

好了，说了这么多，还是看看例子会比较直观。

（1）新的目标的实现，不再需要自己来实现 Subject 定义，在具体实现的时候，也不是继承 Subject 了，而是改成继承 Java 中定义的 Observable。示例代码如下：

```
/**
 * 报纸对象，具体的目标实现
 */
public class NewsPaper extends java.util.Observable {
    /**
     * 报纸的具体内容
     */
    private String content;
    /**
     * 获取报纸的具体内容
     * @return 报纸的具体内容
     */
    public String getContent() {
        return content;
    }
    /**
     * 示意，设置报纸的具体内容，相当于要出版报纸了
     * @param content 报纸的具体内容
     */
```

```
    public void setContent(String content) {
        this.content = content;
        //内容有了，说明又出新报纸了，那就通知所有的读者
        //注意在用Java中的Observer模式的时候，下面这句话不可少
        this.setChanged();
        //然后主动通知，这里用的是推的方式
        this.notifyObservers(this.content);
        //如果用拉的方式，这么调用
        //this.notifyObservers();
    }
}
```

注意这些实现的改变

（2）再看看新的观察者的实现，不是实现自己定义的观察者接口，而是实现由 Java 提供的 Observer 接口。示例代码如下：

```
/**
 * 真正的读者，为了简单就描述一下姓名
 */
public class Reader implements  java.util.Observer  {
/**
     * 读者的姓名
     */
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
public void update(Observable o, Object obj) {
        //这是采用推的方式
        System.out.println(name
                    +"收到报纸了，阅读先。目标推过来的内容是==="+obj);
```

注意这里实现的接口是 Java 提供的

对于要获取推的数据，在目标实现里面调用的时候必须用推的方式，就是带参数那个，否则这里会是 null

```
 //这是获取拉的数据
 System.out.println(name
                +"收到报纸了，阅读它。主动到目标对象去拉的内容是==="
                +((NewsPaper)o).getContent());
```

```
对于要用拉的方式获取数据，在目标实现里面怎么调用都行，有参无参都行，Java 默认会传递目标的实现对象本身。也就是说，Java 实现观察者模式时默认是拉模型，如果你用推模型调用，那么两种方式都可以获取到值，也就是两种方式可以同时使用
    }
}
```

（3）客户端使用。

客户端跟前面的写法没有太大改变，主要在注册阅读者的时候，调用的方法和以前不一样了。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建一个报纸，作为被观察者
        NewsPaper subject = new NewsPaper();
        //创建阅读者，也就是观察者
        Reader reader1 = new Reader();
        reader1.setName("张三");

        Reader reader2 = new Reader();
        reader2.setName("李四");

        Reader reader3 = new Reader();
        reader3.setName("王五");

        //注册阅读者
        subject.addObserver(reader1);
        subject.addObserver(reader2);
        subject.addObserver(reader3);

        //要出报纸啦
        subject.setContent("本期内容是观察者模式");
    }
}
```

测试一下，运行看看结果。运行结果如下所示：

```
王五收到报纸了，阅读它。目标推过来的内容是===本期内容是观察者模式
王五收到报纸了，阅读它。主动到目标对象去拉的内容是===本期内容是观察者模式
李四收到报纸了，阅读它。目标推过来的内容是===本期内容是观察者模式
```

```
李四收到报纸了，阅读它。主动到目标对象去拉的内容是===本期内容是观察者模式
张三收到报纸了，阅读它。目标推过来的内容是===本期内容是观察者模式
张三收到报纸了，阅读它。主动到目标对象去拉的内容是===本期内容是观察者模式
```

然后认真对比自己实现观察者模式和使用 Java 已有的功能来实现观察者模式，看看有什么不同，有什么相同，好好体会一下。

12.3.4 观察者模式的优缺点

观察者模式具有以下优点。

- 观察者模式实现了观察者和目标之间的抽象耦合

  原本目标对象在状态发生改变的时候，需要直接调用所有的观察者对象，但是抽象出观察者接口以后，目标和观察者就只是在抽象层面上耦合了，也就是说目标只是知道观察者接口，并不知道具体的观察者的类，从而实现目标类和具体的观察者类之间解耦。

- 观察者模式实现了动态联动

  所谓联动，就是做一个操作会引起其他相关的操作。由于观察者模式对观察者注册实行管理，那就可以在运行期间，通过动态地控制注册的观察者，来控制某个动作的联动范围，从而实现动态联动。

- 观察者模式支持广播通信

  由于目标发送通知给观察者是面向所有注册的观察者，所以每次目标通知的信息就要对所有注册的观察者进行广播。当然，也可以通过在目标上添加新的功能来限制广播的范围。

  在广播通信的时候要注意一个问题，就是相互广播造成死循环的问题。比如 A 和 B 两个对象互为观察者和目标对象，A 对象发生状态变化，然后 A 来广播信息，B 对象接收到通知后，在处理过程中，使得 B 对象的状态也发生了改变，然后 B 来广播信息，然后 A 对象接到通知后，又触发广播信息……，如此 A 引起 B 变化，B 又引起 A 变化，从而一直相互广播信息，就造成死循环。

观察者模式的缺点是：

- 可能会引起无谓的操作

  由于观察者模式每次都是广播通信，不管观察者需不需要，每个观察者都会被调用 update 方法，如果观察者不需要执行相应处理，那么这次操作就浪费了。其实浪费了还好，最怕引起误更新，那就麻烦了，比如，本应该在执行这次状态更新前把某个观察者删除掉，这样通知的时候就没有这个观察者了，但是现在忘掉了，那么就会引起误操作。

12.3.5　思考观察者模式

> 观察者模式的本质：触发联动。

1. 观察者模式的本质

当修改目标对象的状态的时候，就会触发相应的通知，然后会循环调用所有注册的观察者对象的相应方法，其实就相当于联动调用这些观察者的方法。

而且这个联动还是动态的，可以通过注册和取消注册来控制观察者，因而可以在程序运行期间，通过动态地控制观察者，来变相地实现添加和删除某些功能处理，这些功能就是观察者在 update 的时候执行的功能。

同时目标对象和观察者对象的解耦，又保证了无论观察者发生怎样的变化，目标对象总是能够正确地联动过来。

理解这个本质对我们非常有用，对于我们识别和使用观察者模式有非常重要的意义，尤其是在变形使用的时候。万变不离其宗。

2. 何时选用观察者模式

建议在以下情况中选用观察者模式。

- 当一个抽象模型有两个方面，其中一个方面的操作依赖于另一个方面的状态变化，那么就可以选用观察者模式，将这两者封装成观察者和目标对象，当目标对象变化的时候，依赖于它的观察者对象也会发生相应的变化。这样就把抽象模型的这两个方面分离开了，使得它们可以独立地改变和复用。
- 如果在更改一个对象的时候，需要同时连带改变其他的对象，而且不知道究竟应该有多少对象需要被连带改变，这种情况可以选用观察者模式，被更改的那一个对象很明显就相当于是目标对象，而需要连带修改的多个其他对象，就作为多个观察者对象了。
- 当一个对象必须通知其他的对象，但是你又希望这个对象和其他被它通知的对象是松散耦合的。也就是说这个对象其实不想知道具体被通知的对象。这种情况可以选用观察者模式，这个对象就相当于是目标对象，而被它通知的对象就是观察者对象了。

12.3.6　Swing 中的观察者模式

Java 的 Swing 中到处都是观察者模式的身影，比如大家熟悉的事件处理就是典型的观察者模式的应用。（说明一下：早期的 Swing 事件处理用的是职责链）。

Swing 组件是被观察的目标，而每个实现监听器的类就是观察者，监听器的接口就是观察者的接口，在调用 addXXXListener 方法的时候就相当于注册观察者。

当组件被单击，状态发生改变的时候，就会产生相应的通知，会调用注册的观察者的方法，就是我们所实现的监听器的方法。

延伸

从这里还可以学一招：如何处理一个观察者观察多个目标对象？

你看一个 Swing 的应用程序，作为一个观察者，经常会注册观察多个不同的目标对象，也就是同一类，既实现了按钮组件的事件处理，又实现了文本框组件的事件处理，是怎么做到的呢？

答案就在监听器接口上，这些监听器接口就相当于观察者接口，也就是说一个观察者要观察多个目标对象，只要不同的目标对象使用不同的观察者接口就好了。当然，这些接口里面的方法也不相同，不再都是 update 方法了。这样一来，不同的目标对象通知观察者所调用的方法也就不同了，这样在具体实现观察者的时候，也就实现成不同的方法，自然就区分开了。

12.3.7 简单变形示例——区别对待观察者

首先声明，这里只是举一个非常简单的变形使用的例子，也可算是基本的观察者模式的功能加强，事实上可以有很多很多的变形应用，这也是为什么我们特别强调大家要深入理解每个设计模式，要把握每个模式的本质的原因。

1. 范例需求

这是一个实际系统的简化需求：在一个水质监测系统中有这样一个功能，当水中的杂质为正常的时候，只是通知监测人员做记录；当为轻度污染的时候，除了通知监测人员做记录外，还要通知预警人员，判断是否需要预警；当为中度或者高度污染的时候，除了通知监测人员做记录外，还要通知预警人员，判断是否需要预警，同时还要通知监测部门领导做相应的处理。

2. 解决思路和范例代码

分析上述需求就会发现，对于水质污染这件事情，有可能会涉及到监测员、预警人员、监测部门领导，根据不同的水质污染情况涉及到不同的人员，也就是说，监测员、预警人员、监测部门领导三者是平行的，职责都是处理水质污染，但是处理的范围不一样。

因此很容易套用观察者模式，如果把水质污染的记录当作被观察的目标的话，那么监测员、预警人员和监测部门领导就都是观察者了。

前面学过的观察者模式，当目标通知观察者的时候是全部都通知，但是现在这个需求是不同的情况来让不同的人处理，怎么办呢？

提示

解决的方式通常有两种，一种是目标可以通知，但是观察者不做任何操作，另外一种是在目标里面进行判断，干脆就不通知了。两种实现方式各有千秋，这里选择后面一种方式来示例，这种方式能够统一逻辑控制，并进行观察者的统一分派，有利于业务控制和今后的扩展。

还是看代码吧，会更直观。

（1）先来定义观察者的接口，这个接口跟前面的示例差别也不大，只是新加了访问

观察人员职务的方法。示例代码如下：

```
/**
 * 水质观察者接口定义
 */
public interface WatcherObserver {
    /**
     * 被通知的方法
     * @param subject 传入被观察的目标对象
     */
    public void update(WaterQualitySubject subject);
    /**
     * 设置观察人员的职务
     * @param job 观察人员的职务
     */
    public void setJob(String job);
    /**
     * 获取观察人员的职务
     * @return 观察人员的职务
     */
    public String getJob();
}
```

getter/setter 方法

（2）定义完接口后，来看看观察者的具体实现。示例代码如下：

```
/**
 * 具体的观察者实现
 */
public class Watcher implements WatcherObserver{
    /**
     * 职务
     */
    private String job;
public String getJob() {
        return this.job;
    }
    public void setJob(String job) {
        this.job = job;
    }

    public void update(WaterQualitySubject subject) {
        //这里采用的是拉的方式
```

收到通知后的处理

```
        System.out.println(job+"获取到通知，当前污染级别为："
                                        +subject.getPolluteLevel());
    }
}
```

（3）再来定义目标的父对象，和以前相比有些改变。

- 把父类实现成抽象的，因为在里面要定义抽象的方法。
- 原来通知所有的观察者的方法被去掉了，这个方法现在需要由子类去实现，要按照业务有区别地来对待观察者，得看看是否需要通知观察者。
- 新添加一个水质污染级别的业务方法，这样在观察者获取目标对象数据的时候，就不需要再知道具体的目标对象，也不需要强制造型了。

示例代码如下：

```
/**
 * 定义水质监测的目标对象
 */
public abstract class WaterQualitySubject {
    /**
     * 用来保存注册的观察者对象
     */
    protected List<WatcherObserver> observers =
                            new ArrayList<WatcherObserver>();
    /**
     * 注册观察者对象
     * @param observer 观察者对象
     */
    public void attach(WatcherObserver observer) {
        observers.add(observer);
    }
    /**
     * 删除观察者对象
     * @param observer 观察者对象
     */
    public void detach(WatcherObserver observer) {
        observers.remove(observer);
    }
    /**
     * 通知相应的观察者对象
     */
    public abstract void notifyWatchers();
    /**
```

注意这个方法不再是通知所有的观察者了，现在要按照业务要求去通知

```
     * 获取水质污染的级别
     * @return 水质污染的级别
     */
    public abstract int getPolluteLevel();
}
```

（4）接下来重点看看目标的实现。在目标对象中，添加一个描述污染级别的属性，在判断是否需要通知观察者的时候，不同的污染程度会对应通知不同的观察者。示例代码如下：

```
/**
 * 具体的水质监测对象
 */
public class WaterQuality extends WaterQualitySubject{
    /**
     * 污染的级别，0表示正常，1表示轻度污染，2表示中度污染，3表示高度污染
     */
    private int polluteLevel = 0;
    /**
     * 获取水质污染的级别
     * @return 水质污染的级别
     */
    public int getPolluteLevel() {
        return polluteLevel;
    }
    /**
     * 当监测水质情况后，设置水质污染的级别
     * @param polluteLevel 水质污染的级别
     */
    public void setPolluteLevel(int polluteLevel) {
        this.polluteLevel = polluteLevel;
        //通知相应的观察者
        this.notifyWatchers();    ← 通知相应的观察者
    }
    /**
     * 通知相应的观察者对象
     */
    public void notifyWatchers() {
        //循环所有注册的观察者
        for(WatcherObserver watcher : observers){
                        //开始根据污染级别判断是否需要通知，由这里总控
```

```
            if(this.polluteLevel >= 0){
                //通知监测员做记录
                if("监测人员".equals(watcher.getJob())){
                    watcher.update(this);
                }
            }
            if(this.polluteLevel >= 1){
                //通知预警人员
                if("预警人员".equals(watcher.getJob())){
                    watcher.update(this);
                }
            }
            if(this.polluteLevel >= 2){
                //通知监测部门领导
                if("监测部门领导".equals(
                                    watcher.getJob())){
                    watcher.update(this);
                }
            }
        }
    }
}
```

这里用的是组合逻辑，不能是 else if，用 else if 逻辑就出问题了。另外一个要注意每个判断 pulluteLevel 都是用的“>=”，而不是“==”

（5）大功告成，来写个客户端，测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建水质主题对象
        WaterQuality subject = new WaterQuality();
        //创建几个观察者
        WatcherObserver watcher1 = new Watcher();
        watcher1.setJob("监测人员");
        WatcherObserver watcher2 = new Watcher();
        watcher2.setJob("预警人员");
        WatcherObserver watcher3 = new Watcher();
        watcher3.setJob("监测部门领导");
        //注册观察者
        subject.attach(watcher1);
        subject.attach(watcher2);
        subject.attach(watcher3);
        //填写水质报告
        System.out.println("当水质为正常的时候------------------〉");
```

```
        subject.setPolluteLevel(0);
        System.out.println("当水质为轻度污染的时候---------------〉 ");
        subject.setPolluteLevel(1);
        System.out.println("当水质为中度污染的时候---------------〉 ");
        subject.setPolluteLevel(2);
    }
}
```

（6）运行一下，看看结果，如下：

```
当水质为正常的时候------------------〉
监测人员获取到通知，当前污染级别为：0
当水质为轻度污染的时候---------------〉
监测人员获取到通知，当前污染级别为：1
预警人员获取到通知，当前污染级别为：1
当水质为中度污染的时候---------------〉
监测人员获取到通知，当前污染级别为：2
预警人员获取到通知，当前污染级别为：2
监测部门领导获取到通知，当前污染级别为：2
```

仔细观察上面输出的结果，你会发现，当填写不同的污染级别时，被通知的人员是不同的。但是这些观察者是不知道这些不同的，观察者只是在自己获得通知的时候去执行自己的工作。具体要不要通知，什么时候通知都是目标对象的工作。

12.3.8　相关模式

- 观察者模式和状态模式

  观察者模式和状态模式是有相似之处的。

  观察者模式是当目标状态发生改变时，触发并通知观察者，让观察者去执行相应的操作。而状态模式是根据不同的状态，选择不同的实现，这个实现类的主要功能就是针对状态相应地操作，它不像观察者，观察者本身还有很多其他的功能，接收通知并执行相应处理只是观察者的部分功能。

  当然观察者模式和状态模式是可以结合使用的。观察者模式的重心在触发联动，但是到底决定哪些观察者会被联动，这时就可以采用状态模式来实现了，也可以采用策略模式来进行选择需要联动的观察者。

- 观察者模式和中介者模式

  观察者模式和中介者模式是可以结合使用的。

  前面的例子中目标都只是简单地通知一下，然后让各个观察者自己去完成更新就结束了。如果观察者和被观察的目标之间的交互关系很复杂，比如，有一个界面，里面有三个下拉列表组件，分别是选择国家、省份/州、具体的城市，很明显这是一个三级联动，当你选择一个国家的时候，省份/州应该相应改变数据，

省份/州一改变，具体的城市也需要改变。

这种情况下，很明显需要相关的状态都联动准备好了，然后再一次性地通知观察者。也就是界面做更新处理，不会仅国家改变一下，省份和城市还没有改，就通知界面更新。这种情况就可以使用中介者模式来封装观察者和目标的关系。

在使用 Swing 的小型应用里面，也可以使用中介者模式，比如，把一个界面所有的事件用一个对象来处理，把一个组件触发事件以后，需要操作其他组件的动作都封装到一起，这个对象就是典型的中介者。

第 13 章　命令模式（Command）

13.1 场景问题

13.1.1 如何开机

估计有些朋友看到这个标题会非常奇怪，电脑装配好了，如何开机？不就是按下启动按钮就可以了吗？难道还有什么玄机不成。

对于使用电脑的客户——就是我们来说，开机确实很简单，按下启动按钮，然后耐心等待就可以了。但是当我们按下启动按钮以后，谁来处理？如何处理？都经历了怎样的过程？才让电脑真正地启动起来，供我们使用。

先来简单地认识一下电脑的启动过程。做一个了解即可。

（1）当我们按下启动按钮，电源开始向主板和其他设备供电。

（2）主板的系统 BIOS（基本输入输出系统）开始加电后自检。

（3）主板的 BIOS 会依次去寻找显卡等其他设备的 BIOS，并让它们自检或者初始化。

（4）开始检测 CPU、内存、硬盘、光驱、串口、并口、软驱、即插即用设备等。

（5）BIOS 更新 ESCD（扩展系统配置数据），ESCD 是 BIOS 和操作系统交换硬件配置数据的一种手段。

（6）等前面的事情都完成后，BIOS 才按照用户的配置进行系统引导，进入操作系统里面，等到操作系统装载并初始化完毕，就出现我们熟悉的系统登录界面了。

13.1.2 与我何干

讲了一些电脑启动的过程，有些朋友会想，这与我何干呢？

没错，看起来这些硬件知识跟你没有什么大的关系，但是，如果现在提出一个要求：请你用软件把上面的过程表现出来，你该如何实现？

首先把上面的过程总结一下。主要有几个步骤：首先加载电源，然后是设备检查，接下来是装载系统，最后电脑就正常启动了。可是谁来完成这些过程？如何完成呢？

不能让使用电脑的客户——就是我们来做这些工作吧，真正完成这些工作的是主板。那么客户和主板如何发生联系呢？现实中，是用连接线把按钮连接到主板上的，这样当客户按下按钮的时候，就相当于发命令给主板，让主板去完成后续的工作。

另外，从客户的角度来看，开机就是按下按钮，不管什么样的主板都是一样的。也就是说，客户只管发出命令，谁接收命令？谁实现命令？如何实现？客户是不关心的。

13.1.3 有何问题

把上面的问题抽象描述一下：客户端只是想要发出命令或者请求，不关心请求的真正接收者是谁，也不关心具体如何实现，而且同一个请求的动作可以有不同的请求内容，当然具体的处理功能也不一样，请问该怎样实现？

13.2 解决方案

13.2.1 使用命令模式来解决问题

用来解决上述问题的一个合理的解决方案就是命令模式。那么什么是命令模式呢？

1. 命令模式的定义

> 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可撤销的操作。

2. 应用命令模式来解决问题的思路

下面来看看实际电脑的解决方案。

先画个图来描述一下，看看实际的电脑是如何处理上面描述的这个问题的，如图 13.1 所示。

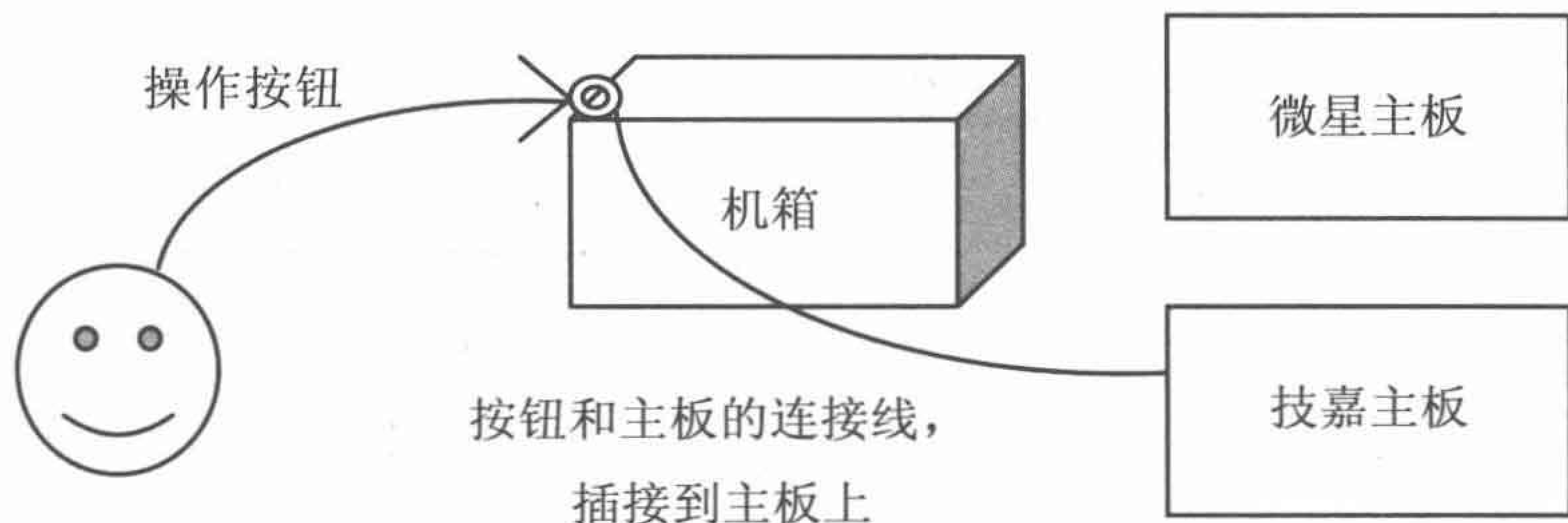


图 13.1　电脑操作示意图

当客户按下按钮的时候，按钮本身并不知道如何处理，于是通过连接线来请求主板，让主板去完成真正启动机器的功能。

这里为了描述它们之间的关系，把主板画到了机箱的外面。如果连接线连接到不同的主板，那么真正执行按钮请求的主板也就不同了，而客户是不知道这些变化的。

通过引入按钮和连接线，来让发出命令的客户和命令的真正实现者——主板完全解耦，客户操作的始终是按钮，按钮后面的事情客户就统统不管了。

要用程序来解决上面提出的问题，**一种自然的方案就是来模拟上述解决思路。**

在命令模式中，会定义一个命令的接口，用来约束所有的命令对象，然后提供具体的命令实现，每个命令实现对象是对客户端某个请求的封装，对应于机箱上的按钮，一个机箱上可以有很多按钮，也就相当于会有多个具体的命令实现对象。

在命令模式中，命令对象并不知道如何处理命令，会有相应的接收者对象来真正执行命令。就像电脑的例子，机箱上的按钮并不知道如何处理功能，而是把这个请求转发给主板，由主板来执行真正的功能，这个主板就相当于命令模式的接收者。

在命令模式中，命令对象和接收者对象的关系，并不是与生俱来的，需要有一个装

配的过程，命令模式中的 Client 对象可以实现这样的功能。这就相当于在电脑的例子中，有了机箱上的按钮，也有了主板，还需要有一个连接线把这个按钮连接到主板上才行。

命令模式还会提供一个 Invoker 对象来持有命令对象。就像电脑的例子，机箱上会有多个按钮，这个机箱就相当于命令模式的 Invoker 对象。这样一来，命令模式的客户端就可以通过 Invoker 来触发并要求执行相应的命令了，这也相当于真正的客户是按下机箱上的按钮来操作电脑一样。

13.2.2 命令模式的结构和说明

命令模式的结构如图 13.2 所示：

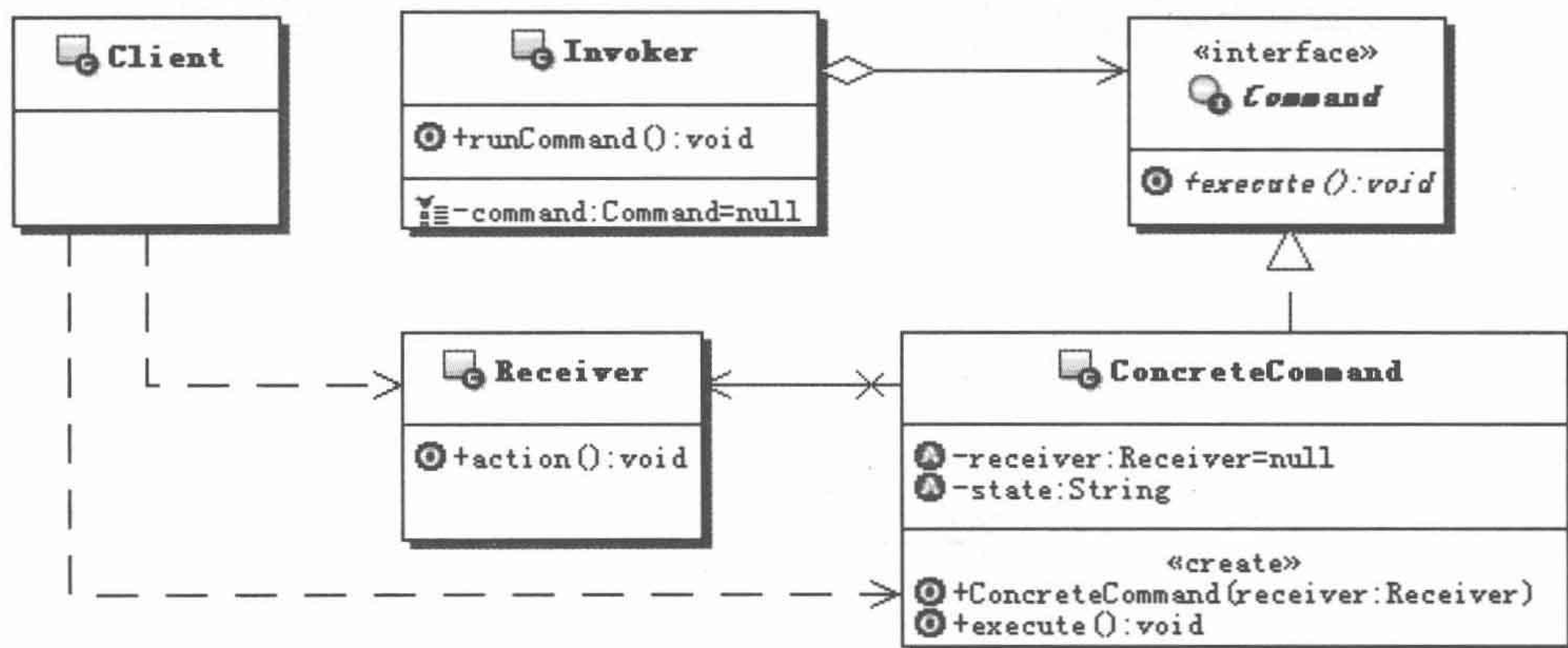


图 13.2　命令模式的结构图

- Command：定义命令的接口，声明执行的方法。
- ConcreteCommand：命令接口实现对象，是“虚”的实现；通常会持有接收者，并调用接收者的功能来完成命令要执行的操作。
- Receiver：接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能。
- Invoker：要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。这个是客户端真正触发命令并要求命令执行相应操作的地方，也就是说相当于使用命令对象的入口。
- Client：创建具体的命令对象，并且设置命令对象的接收者。注意这个不是我们常规意义上的客户端，而是在组装命令对象和接收者，或许，把这个 Client 称为装配者会更好理解，因为真正使用命令的客户端是从 Invoker 来触发执行。

13.2.3 命令模式示例代码

（1）先来看看命令接口的定义。示例代码如下：

```
/**
 * 命令接口，声明执行的操作
```

```
 */
public interface Command {
    /**
     * 执行命令对应的操作
     */
    public void execute();
}
```

（2）接下来看看具体的命令实现对象。示例代码如下：

```
/**
 * 具体的命令实现对象
 */
public class ConcreteCommand implements Command {
    /**
     * 持有相应的接收者对象
     */
    private Receiver receiver = null;
    /**
     * 示意，命令对象可以有自己的状态
     */
    private String state;
    /**
     * 构造方法，传入相应的接收者对象
     * @param receiver 相应的接收者对象
     */
    public ConcreteCommand(Receiver receiver){
        this.receiver = receiver;
    }
    public void execute() {
        //通常会转调接收者对象的相应方法，让接收者来真正执行功能
        receiver.action();
    }
}
```

（3）再来看看接收者对象的实现示意。示例代码如下：

```
/**
 * 接收者对象
 */
public class Receiver {
    /**
     * 示意方法，真正执行命令相应的操作
```

```
     */
    public void action(){
        //真正执行命令操作的功能代码
    }
}
```

（4）下面来看看 Invoker 对象。示例代码如下：

```
/**
 * 调用者
 */
public class Invoker {
    /**
     * 持有命令对象
     */
    private Command command = null;
    /**
     * 设置调用者持有的命令对象
     * @param command 命令对象
     */
    public void setCommand(Command command) {
        this.command = command;
    }
    /**
     * 示意方法，要求命令执行请求
     */
    public void runCommand() {
        //调用命令对象的执行方法
        command.execute();
    }
}
```

（5）再来看看 Client 的实现。

> **注意** **注意这个不是我们通常意义上的测试客户端，主要功能是要创建命令对象并设定它的接收者，因此这里并没有调用执行的代码。**

示例代码如下：

```
public class Client {
    /**
     * 示意，负责创建命令对象，并设定它的接收者
     */
    public void assemble(){
```

```
        //创建接收者
        Receiver receiver = new Receiver();
        //创建命令对象，设定它的接收者
        Command command = new ConcreteCommand(receiver);
        //创建Invoker，把命令对象设置进去
        Invoker invoker = new Invoker();
        invoker.setCommand(command);
    }
}
```

13.2.4　使用命令模式来实现示例

要使用命令模式来实现示例，需要先把命令模式中所涉及的各个部分，在实际的示例中对应出来，然后才能按照命令模式的结构来设计和实现程序。根据前面描述的解决思路，大致对应如下：

- 机箱上的按钮就相当于是命令对象。
- 机箱相当于是 Invoker。
- 主板相当于接收者对象。
- 命令对象持有一个接收者对象，就相当于是给机箱的按钮连上了一根连接线。
- 当机箱上的按钮被按下的时候，机箱就把这个命令通过连接线发送出去。

主板类才是真正实现开机功能的地方，是真正执行命令的地方，也就是“接收者”。命令的实现对象，其实是个“虚”的实现，就如同那根连接线，它哪知道如何实现啊，还不就是把命令传递给连接线连到的主板。

使用命令模式来实现示例的结构如图 13.3 所示。

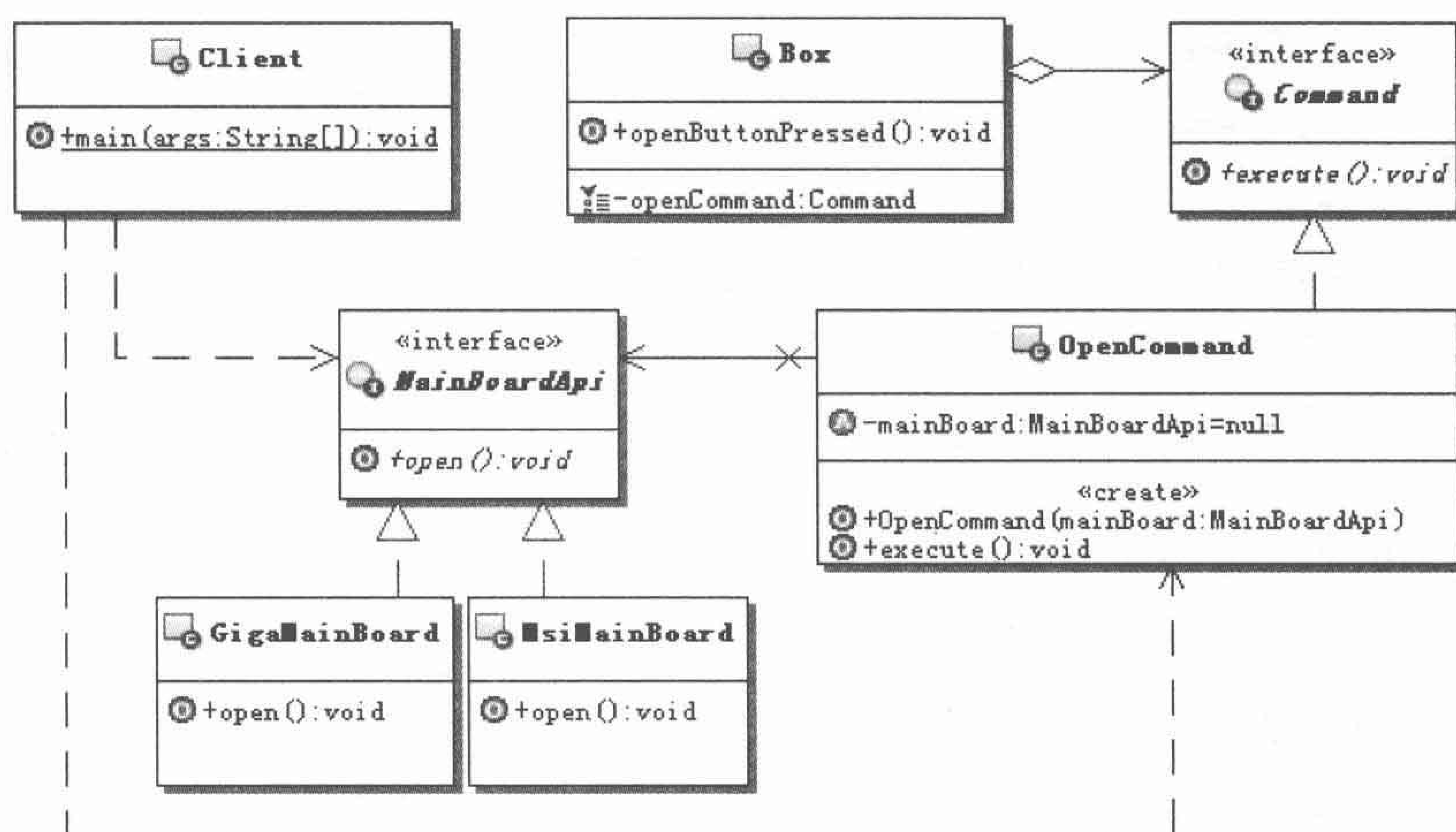


图 13.3　使用命令模式来实现示例的结构示意图

还是来看看示例代码，会比较清楚。

1．定义主板

根据前面的描述，我们会发现，真正执行客户命令或请求的是主板，也只有主板才知道如何去实现客户的命令，因此先来抽象主板，把它用对象描述出来。

先来定义主板的接口，最起码主板会有一个能开机的方法。示例代码如下：

```
/**
 * 主板的接口
 */
public interface MainBoardApi {
    /**
     * 主板具有能开机的功能
     */
    public void open();
}
```

定义了接口，那就接着定义实现类吧，定义两个主板的实现类，一个是技嘉主板，一个是微星主板，现在的实现是一样的，但是不同的主板对同一个命令的操作可以是不同的，这点大家要注意。由于两个实现基本一样，就示例一个。示例代码如下：

```
/**
 * 技嘉主板类，开机命令的真正实现者，在Command模式中充当Receiver
 */
public class GigaMainBoard implements MainBoardApi{
    /**
     * 真正的开机命令的实现
     */
    public void open(){
        System.out.println("技嘉主板现在正在开机，请等候");
        System.out.println("接通电源......");
        System.out.println("设备检查......");
        System.out.println("装载系统......");
        System.out.println("机器正常运转起来......");
        System.out.println("机器已经正常打开，请操作");
    }
}
```

微星主板的实现和这个完全一样，只是把技嘉改名成微星即可。

2. **定义命令接口和命令的实现**

对于客户来说，开机就是按下按钮，别的什么都不想做。把用户的这个动作抽象一下，就相当于客户发出了一个命令或者请求，其他的客户就不关心了。为描述客户的命令，现在定义出一个命令的接口，里面只有一个方法，那就是执行。示例代码如下：

```
/**
 * 命令接口，声明执行的操作
```

```
 */
public interface Command {
    /**
     * 执行命令对应的操作
     */
    public void execute();
}
```

有了命令的接口，再来定义一个具体的实现，其实就是模拟现实中机箱上按钮的功能。因为我们按下的是按钮，但是按钮本身是不知道如何启动电脑的，它需要把这个命令转给主板，让主板去真正地执行开机功能。示例代码如下：

```
/**
 * 开机命令的实现，实现Command接口
 * 持有开机命令的真正实现，通过调用接收者的方法来实现命令
 */
public class OpenCommand implements Command{
    /**
     * 持有真正实现命令的接收者——主板对象
     */
    private MainBoardApi mainBoard = null;
    /**
     * 构造方法，传入主板对象
     * @param mainBoard 主板对象
     */
    public OpenCommand(MainBoardApi mainBoard) {
        this.mainBoard = mainBoard;
    }

    public void execute() {
        //对于命令对象，根本不知道如何开机，会转调主板对象
        //让主板去完成开机的功能
        this.mainBoard.open();
    }
}
```

由于客户不想直接和主板打交道，而且客户根本不知道具体的主板是什么，客户只是希望按下启动按钮，电脑就正常启动了，就这么简单。就算换了主板，客户还是一样地按下启动按钮就可以了。

换句话说就是：客户想要和主板完全解耦，怎么办呢？

这就需要在客户和主板之间建立一个中间对象。客户发出的命令传递给这个中间对象，然后由这个中间对象去找真正的执行者——主板，来完成工作。

很显然，这个中间对象就是上面的命令实现对象。

注意	**这个实现其实是个虚的实现，真正的实现是主板完成的，在这个虚的实现里面，是通过转调主板的功能来实现的，主板对象实例，是从外面传进来的。**

3. 提供机箱

客户需要操作按钮，按钮是放置在机箱之上的，所以需要把机箱也定义出来。示例代码如下：

```java
/**
 * 机箱对象，本身有按钮，持有按钮对应的命令对象
 */
public class Box {
    /**
     * 开机命令对象
     */
    private Command openCommand;
    /**
     * 设置开机命令对象
     * @param command 开机命令对象
     */
    public void setOpenCommand(Command command){
        this.openCommand = command;
    }
    /**
     * 提供给客户使用，接收并响应用户请求，相当于按钮被按下触发的方法
     */
    public void openButtonPressed(){
        //按下按钮，执行命令
        openCommand.execute();
    }
}
```

4. 客户使用按钮

抽象好了机箱和主板，命令对象也准备好了，客户想要使用按钮来完成开机的功能。在使用之前，客户的第一件事情就应该是把按钮和主板组装起来，形成一个完整的机器。

在实际生活中，是由装机工程师来完成这部分工作。这里为了测试简单，直接写在客户端开头了。机器组装好过后，客户应该把与主板连接好的按钮对象放置到机箱上，等待客户随时操作。把这个过程也用代码描述出来，示例代码如下：

```java
public class Client {
    public static void main(String[] args) {
```

```
        //1：把命令和真正的实现组合起来，相当于在组装机器
        //把机箱上按钮的连接线插接到主板上
        MainBoardApi mainBoard = new GigaMainBoard();
        OpenCommand openCommand = new OpenCommand(mainBoard);
        //2：为机箱上的按钮设置对应的命令，让按钮知道该干什么
        Box box = new Box();
        box.setOpenCommand(openCommand);

        //3：然后模拟按下机箱上的按钮
        box.openButtonPressed();
    }
}
```

运行一下，看看效果。输出如下：

```
技嘉主板现在正在开机，请等候
接通电源......
设备检查......
装载系统......
机器正常运转起来......
机器已经正常打开，请操作
```

你可以给命令对象组装不同的主板实现类，然后再次测试，看看效果。

事实上，你会发现，如果对象结构已经组装好了以后，对于真正的客户端，也就是真实的用户而言，任务就是面对机箱，按下机箱上的按钮就可以执行开机的命令了，实际生活中也是这样的。

5. 小结

如同前面的示例，把客户的开机请求封装成为一个 OpenCommand 对象，客户的开机操作就变成了执行 OpenCommand 对象的方法了？如果还有其他的命令对象，比如让机器重启的 ResetCommand 对象。那么客户按下按钮的动作，就可以用这不同的命令对象去匹配，也就是对客户进行参数化。

用大白话描述就是：客户按下一个按钮，到底是开机还是重启，那要看参数化配置的是哪一个具体的按钮对象，如果参数化的是开机的命令对象，那就执行开机的功能，如果参数化的是重启的命令对象，那就执行重启的功能。虽然按下的是同一个按钮，但是请求是不同的，对应执行的功能也就不同了。

> 提示　**在模式讲解的时候会给大家一个参数化配置的示例，这里就不多讲了。至于对请求排队或记录请求日志，以及支持可撤销的操作等功能，也放到模式讲解一节里面。**

13.3 模式讲解

13.3.1 认识命令模式

1. 命令模式的关键

命令模式的关键之处就是把请求封装成为对象，也就是命令对象，并定义了统一的执行操作的接口，这个命令对象可以被存储、转发、记录、处理、撤销等，整个命令模式都是围绕这个对象在进行。

2. 命令模式的组装和调用

在命令模式中经常会有一个命令的组装者，用它来维护命令的“虚”实现和真实实现之间的关系。如果是超级智能的命令，也就是说命令对象自己完全实现好了，不需要接收者，那就是命令模式的退化，不需要接收者，自然也不需要组装者了。

而真正的用户就是具体化请求的内容，然后提交请求进行触发就可以了。真正的用户会通过 Invoker 来触发命令。

在实际开发过程中，Client 和 Invoker 可以融合在一起，由客户在使用命令模式的时候，先进行命令对象和接收者的组装，组装完成后，就可以调用命令执行请求。

3. 命令模式的接收者

接收者可以是任意的类，对它没有什么特殊要求，这个对象知道如何真正执行命令的操作，执行时是从 Command 的实现类里面转调过来。

一个接收者对象可以处理多个命令，接收者和命令之间没有约定的对应关系。接收者提供的方法个数、名称、功能和命令中的可以不一样，只要能够通过调用接收者的方法来实现命令对应的功能就可以了。

4. 智能命令

在标准的命令模式里面，命令的实现类是没有真正实现命令要求的功能的，真正执行命令的功能的是接收者。

如果命令的实现对象比较智能，它自己就能真实地实现命令要求的功能，而不再需要调用接收者，那么这种情况就称为智能命令。

也可以有半智能的命令，命令对象知道部分实现，其他的还是需要调用接收者来完成，也就是说命令的功能由命令对象和接收者共同来完成。

5. 发起请求的对象和真正实现的对象是解耦的

请求究竟由谁处理？如何处理？发起请求的对象是不知道的，也就是发起请求的对象和真正实现的对象是解耦的。发起请求的对象只管发出命令，其他的就不管了。

6. 命令模式的调用顺序示意图

使用命令模式的过程分成两个阶段，一个阶段是组装命令对象和接收者对象的过程；另外一个阶段是触发调用 Invoker，来让命令真正执行的过程。

先看看组装过程的调用顺序示意图，如图 13.4 所示。

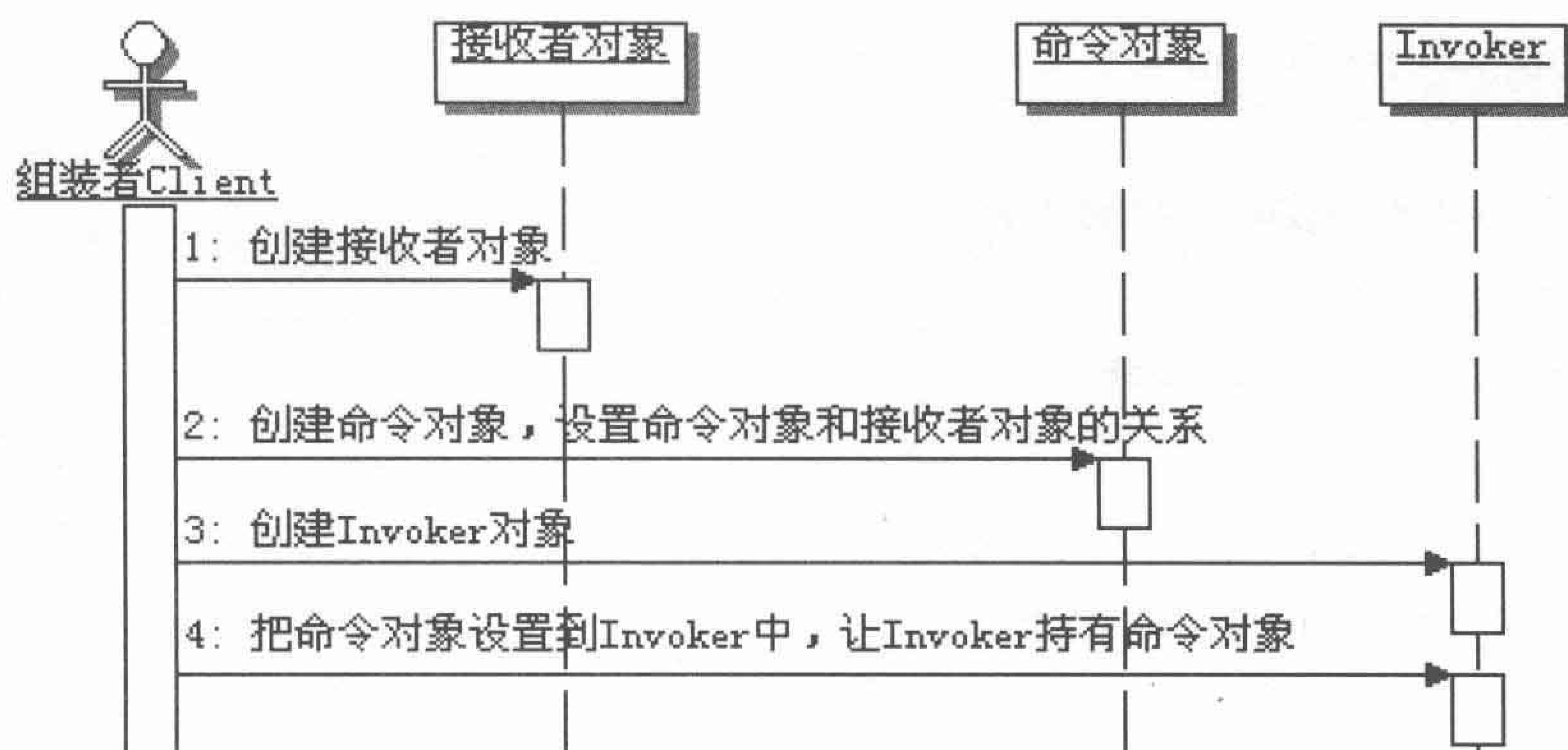


图 13.4　命令模式组装过程的调用顺序示意图

接下来再看看真正执行命令时的调用顺序示意图，如图 13.5 所示。

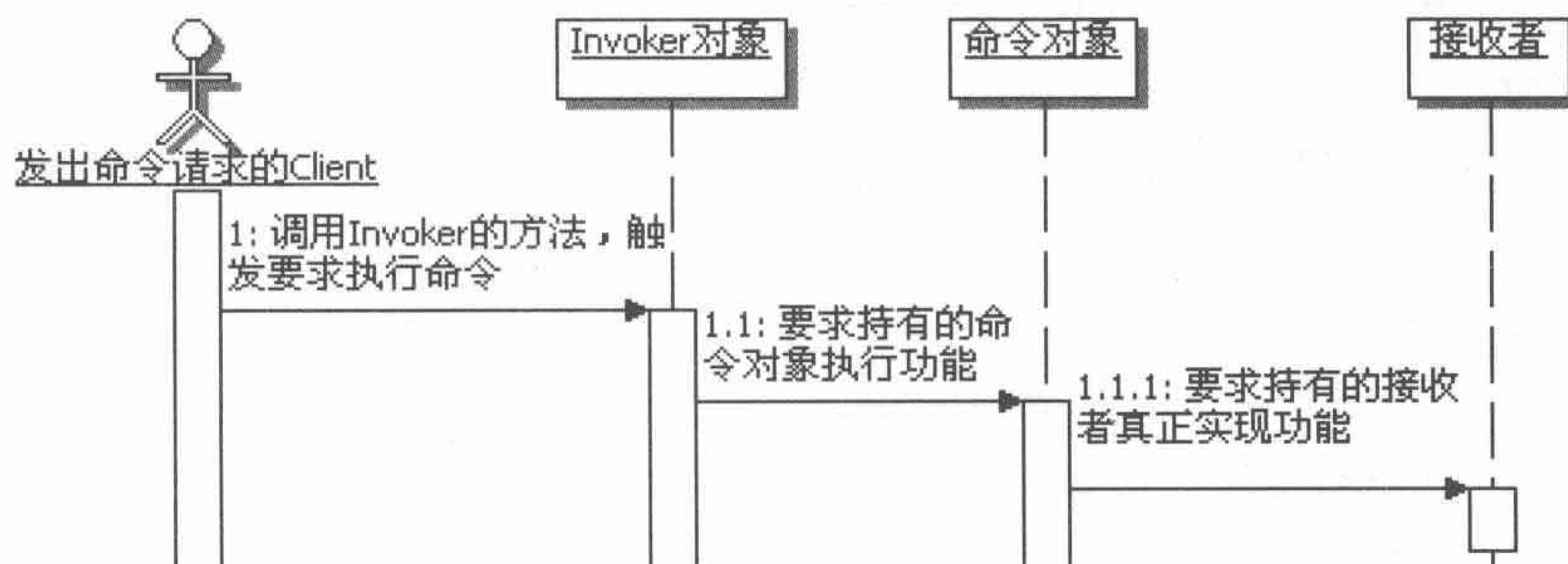


图 13.5　命令模式执行过程的调用顺序示意图

13.3.2　参数化配置

所谓命令模式的参数化配置，指的是：**可以用不同的命令对象，去参数化配置客户的请求。**

像前面描述的那样：客户按下一个按钮，到底是开机还是重启，那要看参数化配置的是哪一个具体的按钮对象。如果参数化的是开机的命令对象，那就执行开机的功能；如果参数化的是重启的命令对象，那就执行重启的功能。虽然按下的是同一个按钮，相当于是同一个请求，但是为请求配置不同的按钮对象，就会执行不同的功能。

把这个功能用代码实现出来，一起来体会一下命令模式的参数化配置。

（1）先定义主板接口。现在想要添加一个重启的按钮，因此主板需要添加一个方法来实现重启的功能。示例代码如下：

```
/**
 * 主板的接口
 */
public interface MainBoardApi {
    /**
     * 主板具有能开机的功能
     */
```

```
    public void open();
    /**
     * 主板具有实现重启的功能
     */
    public void reset();
}
```

新添加的实现重启机器的功能

接口发生了改变，实现类也需要相应地改变。由于两个主板的实现示意差不多，因此这里只示例一个。示例代码如下：

```
/**
 * 技嘉主板类，命令的真正实现者，在Command模式中充当Receiver
 */
public class GigaMainBoard implements MainBoardApi{
    /**
     * 真正的开机命令的实现
     */
    public void open(){
        System.out.println("技嘉主板现在正在开机，请等候");
        System.out.println("接通电源......");
        System.out.println("设备检查......");
        System.out.println("装载系统......");
        System.out.println("机器正常运转起来......");
        System.out.println("机器已经正常打开，请操作");
    }
    /**
     * 真正的重新启动机器命令的实现
     */
    public void reset(){
        System.out.println("技嘉主板现在正在重新启动机器，请等候");
        System.out.println("机器已经正常打开，请操作");
    }
}
```

（2）接下来定义命令和按钮。命令接口没有任何变化，原有的开机命令的实现也没有任何变化，只是新添加了一个重启命令的实现。示例代码如下：

```
/**
 * 重启机器命令的实现，实现Command接口
 * 持有重启机器命令的真正实现，通过调用接收者的方法来实现命令
 */
public class ResetCommand implements Command{
    /**
```

```
     * 持有真正实现命令的接收者——主板对象
     */
    private MainBoardApi mainBoard = null;
    /**
     * 构造方法，传入主板对象
     * @param mainBoard 主板对象
     */
    public ResetCommand(MainBoardApi mainBoard) {
        this.mainBoard = mainBoard;
    }

    public void execute() {
        //对于命令对象，根本不知道如何重启机器，会转调主板对象
        //让主板去完成重启机器的功能
        this.mainBoard.reset();
    }
}
```

（3）持有命令的机箱也需要修改。现在不只一个命令按钮了，有两个了，所以需要在机箱类里面新添加重启的按钮，为了简单，没有做成集合。示例代码如下：

```
/**
 * 机箱对象，本身有按钮，持有按钮对应的命令对象
 */
public class Box {
    private Command openCommand;
    public void setOpenCommand(Command command){
        this.openCommand = command;
    }
    public void openButtonPressed(){        // 原有的开机命令
        //按下按钮，执行命令
        openCommand.execute();
    }
    /**
     * 重启机器命令对象
     */
    private Command resetCommand;
    /**
     * 设置重启机器命令对象
     * @param command
     */
    public void setResetCommand(Command command){
```

```
        this.resetCommand = command;
    }
    /**
     * 提供给客户使用，接收并响应用户请求，相当于重启按钮被按下触发的方法
     */
    public void resetButtonPressed(){
        //按下按钮，执行命令
        resetCommand.execute();
    }
}
```

（4）看看客户如何使用这两个按钮。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //1：把命令和真正的实现组合起来，相当于在组装机器
        //把机箱上按钮的连接线插接到主板上
        MainBoardApi mainBoard = new GigaMainBoard();
        //创建开机命令
        OpenCommand openCommand = new OpenCommand(mainBoard);
        //创建重启机器的命令
        ResetCommand resetCommand = new ResetCommand(mainBoard);
        //2：为机箱上的按钮设置对应的命令，让按钮知道该干什么
        Box box = new Box();
        //先正确配置，就是开机按钮对开机命令，重启按钮对重启命令
        box.setOpenCommand(openCommand);
        box.setResetCommand(resetCommand);

        //3：模拟按下机箱上的按钮
        System.out.println("正确配置下------------------------->");
        System.out.println(">>>按下开机按钮：>>>");
        box.openButtonPressed();
        System.out.println(">>>按下重启按钮：>>>");
        box.resetButtonPressed();

        //然后来错误配置一回，反正是进行参数化配置
        //就是开机按钮对重启命令，重启按钮对开机命令
        box.setOpenCommand(resetCommand);
        box.setResetCommand(openCommand);
        //4：再来模拟按下机箱上的按钮
        System.out.println("错误配置下------------------------->");
        System.out.println(">>>按下开机按钮：>>>");
```

```
        box.openButtonPressed();
        System.out.println(">>>按下重启按钮：>>>");
        box.resetButtonPressed();
    }
}
```

运行一下看看，很有意思。结果如下：

```
正确配置下------------------------->
>>>按下开机按钮：>>>
技嘉主板现在正在开机，请等候
接通电源......
设备检查......
装载系统......
机器正常运转起来......
机器已经正常打开，请操作
>>>按下重启按钮：>>>
技嘉主板现在正在重新启动机器，请等候
机器已经正常打开，请操作
错误配置下------------------------->
>>>按下开机按钮：>>>
技嘉主板现在正在重新启动机器，请等候
机器已经正常打开，请操作
>>>按下重启按钮：>>>
技嘉主板现在正在开机，请等候
接通电源......
设备检查......
装载系统......
机器正常运转起来......
机器已经正常打开，请操作
```


13.3.3　可撤销的操作

可撤销操作的意思就是：放弃该操作，回到未执行该操作前的状态。这个功能是一个非常重要的功能，几乎所有的 GUI 应用中都有撤销操作的功能。GUI 的菜单是命令模式最典型的应用之一，所以总是能在菜单上找到撤销这样的菜单项。

既然这么常用，那该如何实现呢？

有两种基本的思路来实现可撤销的操作，**一种是补偿式，又称反操作式**，比如被撤销的操作是加的功能，那撤销的实现就变成减的功能；同理被撤销的操作是打开的功能，那么撤销的实现就变成关闭的功能。

另外一种方式是**存储恢复式**，意思就是把操作前的状态记录下来，然后要撤销操作

的时候就直接恢复回去就可以了。

> 提示 **这里先讲第一种方式，就是补偿式或者反操作式，第二种方式放到备忘录模式中进行讲解，详见19.3.4。**

为了让大家更好地理解可撤销操作的功能，还是用一个例子来说明会比较清楚。

1. 范例需求

考虑一个计算器的功能，最简单的那种，只能实现加减法运算，现在要让这个计算器支持可撤销的操作。

2. 补偿式或者反操作式的解决方案

（1）在实现命令接口之前，先来定义真正实现计算的接口，没有它命令就什么都做不了，操作运算的接口的示例代码如下：

```
/**
 * 操作运算的接口
 */
public interface OperationApi {
    /**
     * 获取计算完成后的结果
     * @return 计算完成后的结果
     */
    public int getResult();
    /**
     * 设置计算开始的初始值
     * @param result 计算开始的初始值
     */
    public void setResult(int result);
    /**
     * 执行加法
     * @param num 需要加的数
     */
    public void add(int num);
    /**
     * 执行减法
     * @param num 需要减的数
     */
    public void substract(int num);
}
```

定义了接口，再来看看真正执行加减法的实现。示例代码如下：

```java
/**
 * 运算类，真正实现加减法运算
 */
public class Operation implements OperationApi{
    /**
     * 记录运算的结果
     */
    private int result;
    public int getResult() {
        return result;
    }
    public void setResult(int result) {
        this.result = result;
    }
    public void add(int num){
        //实现加法功能
        result += num;
    }
    public void substract(int num){
        //实现减法功能
        result -= num;
    }
}
```


（2）接下来来抽象命令接口。由于要支持可撤销的功能，所以除了和前面一样定义一个执行方法外，还需要定义一个撤销操作的方法。示例代码如下：

```java
/**
 * 命令接口，声明执行的操作，支持可撤销操作
 */
public interface Command {
    /**
     * 执行命令对应的操作
     */
    public void execute();
    /**
     * 执行撤销命令对应的操作
     */
    public void undo();
}
```

（3）应该来实现命令了，具体的命令分成了加法命令和减法命令。

先来看看加法命令的实现。示例代码如下：

```
/**
 * 具体的加法命令实现对象
 */
public class AddCommand implements Command{
    /**
     * 持有具体执行计算的对象
     */
    private OperationApi operation = null;
    /**
     * 操作的数据，也就是要加上的数据
     */
    private int opeNum;

    public void execute() {
        //转调接收者去真正执行功能，这个命令是做加法
        this.operation.add(opeNum);
    }
    public void undo() {
        //转调接收者去真正执行功能
        //命令本身是做加法，那么撤销的时候就是做减法了
        this.operation.substract(opeNum);
    }
/**
     * 构造方法，传入具体执行计算的对象
     * @param operation 具体执行计算的对象
     * @param opeNum 要加上的数据
     */
    public AddCommand(OperationApi operation,int opeNum){
        this.operation = operation;
        this.opeNum = opeNum;
    }
}
```

转调接收者的功能，尤其是 undo 方法，是执行方法的反向操作

减法命令和加法类似，只是在实现的时候和加法反过来了。示例代码如下：

```
/**
 * 具体的减法命令实现对象
 */
public class SubstractCommand implements Command{
    /**
```

```
     * 持有具体执行计算的对象
     */
    private OperationApi operation = null;
    /**
     * 操作的数据，也就是要减去的数据
     */
    private int opeNum;
    /**
     * 构造方法，传入具体执行计算的对象
     * @param operation 具体执行计算的对象
     * @param opeNum 要减去的数据
     */
    public SubstractCommand(OperationApi operation,int opeNum){
        this.operation = operation;
        this.opeNum = opeNum;
    }

    public void execute() {
        //转调接收者去真正执行功能，这个命令是做减法
        this.operation.substract(opeNum);
    }
    public void undo() {
        //转调接收者去真正执行功能
        //命令本身是做减法，那么撤销的时候就是做加法了
        this.operation.add(opeNum);
    }
}
```

（4）接下来应该看看计算器了。计算器就相当于 Invoker，持有多个命令对象，计算器是实现可撤销操作的地方。

为了大家更好地理解可撤销的功能，先来看看不加可撤销操作的计算器类是什么样子，然后再添加上可撤销的功能示例。示例代码如下：

```
/**
 * 计算器类，计算器上有加法按钮、减法按钮
 */
public class Calculator {
    /**
     * 持有执行加法的命令对象
     */
    private Command addCmd = null;
```

```
    /**
     * 持有执行减法的命令对象
     */
    private Command substractCmd = null;
    /**
     * 设置执行加法的命令对象
     * @param addCmd 执行加法的命令对象
     */
    public void setAddCmd(Command addCmd) {
        this.addCmd = addCmd;
    }
    /**
     * 设置执行减法的命令对象
     * @param substractCmd 执行减法的命令对象
     */
    public void setSubstractCmd(Command substractCmd) {
        this.substractCmd = substractCmd;
    }
    /**
     * 提供给客户使用，执行加法功能
     */
    public void addPressed(){
        this.addCmd.execute();
    }
    /**
     * 提供给客户使用，执行减法功能
     */
    public void substractPressed(){
        this.substractCmd.execute();
    }
}
```

目前看起来同前面的例子实现起来差不多，现在就在这个基本的实现上来添加可撤销操作的功能。

要想实现可撤销操作，首先就需要把操作过的命令记录下来，形成命令的历史列表，撤销的时候就从最后一个开始执行撤销。因此我们先在计算器类里面加上命令历史列表，示例代码如下：

```
/**
* 命令的操作的历史记录，在撤销的时候用
```

```
*/
private List<Command> undoCmds = new ArrayList<Command>();
```

什么时候向命令的历史记录里面加值呢？

很简单，答案是在每个操作按钮被按下的时候，也就是你操作加法按钮或者减法按钮的时候。示例代码如下：

```
public void addPressed(){
        this.addCmd.execute();
        //把操作记录到历史记录里面
        undoCmds.add(this.addCmd);
    }
    public void substractPressed(){
        this.substractCmd.execute();
        //把操作记录到历史记录里面
        undoCmds.add(this.substractCmd);
    }
```

然后在计算器类里面添加上一个撤销的按钮，如果它被按下，那么就从命令历史记录里取出最后一个命令来撤销，撤销完成后要把已经撤销的命令从历史记录里面删除掉，相当于没有执行过该命令。

示例代码如下：

```
public void undoPressed(){
    if(this.undoCmds.size()>0){
        //取出最后一个命令来撤销
        Command cmd = this.undoCmds.get(this.undoCmds.size()-1);
        cmd.undo();
        //然后把最后一个命令删除掉
        this.undoCmds.remove(cmd);
    }else{
        System.out.println("很抱歉，没有可撤销的命令");
    }
}
```

同样的方式，还可以实现恢复的功能。也为恢复设置一个可恢复的列表，需要恢复的时候从列表里面取最后一个命令进行重新执行就可以了。示例代码如下：

```
/**
* 命令被撤销的历史记录，在恢复时用
*/
private List<Command> redoCmds = new ArrayList<Command>();
```

那么什么时候向这个集合里面赋值呢？大家要注意，恢复的命令数据是来源于撤销的命令，也就是说有撤销才会有恢复，所以在撤销的时候向这个集合里面赋值，注意要

在撤销的命令被删除前赋值。示例代码如下，注意蓝色的代码：

```
public void undoPressed(){
    if(this.undoCmds.size()>0){
        //取出最后一个命令来撤销
        Command cmd = this.undoCmds.get(this.undoCmds.size()-1);
        cmd.undo();
        //如果还有恢复的功能，那就把这个命令记录到恢复的历史记录里面
        this.redoCmds.add(cmd);
        //然后把最后一个命令删除掉，
        this.undoCmds.remove(cmd);
    }else{
        System.out.println("很抱歉，没有可撤销的命令");
    }
}
```

那么如何实现恢复呢？请看示例代码：

```
public void redoPressed(){
    if(this.redoCmds.size()>0){
        //取出最后一个命令来重做
        Command cmd = this.redoCmds.get(this.redoCmds.size()-1);
        cmd.execute();
        //把这个命令记录到可撤销的历史记录里面
        this.undoCmds.add(cmd);
        //然后把最后一个命令删除掉
        this.redoCmds.remove(cmd);
    }else{
        System.out.println("很抱歉，没有可恢复的命令");
    }
}
```

好了，上面分步讲解了计算器类，下面一起来看看完整的计算器类的代码：

```
/**
 * 计算器类，计算器上有加法按钮、减法按钮，还有撤销和恢复按钮
 */
public class Calculator {
    /**
     * 命令的操作的历史记录，在撤销时用
     */
    private List<Command> undoCmds = new ArrayList<Command>();
    /**
     * 命令被撤销的历史记录，在恢复时用
```

```
 */
private List<Command> redoCmds = new ArrayList<Command>();

private Command addCmd = null;
private Command substractCmd = null;
public void setAddCmd(Command addCmd) {
    this.addCmd = addCmd;
}
public void setSubstractCmd(Command substractCmd) {
    this.substractCmd = substractCmd;
}
public void addPressed(){
    this.addCmd.execute();
    //把操作记录到历史记录里面
    undoCmds.add(this.addCmd);
}
public void substractPressed(){
    this.substractCmd.execute();
    //把操作记录到历史记录里面
    undoCmds.add(this.substractCmd);
}
public void undoPressed(){
    if(this.undoCmds.size()>0){
        //取出最后一个命令来撤销
        Command cmd = this.undoCmds.get(undoCmds.size()-1);
        cmd.undo();
        //如果还有恢复的功能，那就把这个命令记录到恢复的历史记录里面
        this.redoCmds.add(cmd );
        //然后把最后一个命令删除掉
        this.undoCmds.remove(cmd);
    }else{
        System.out.println("很抱歉，没有可撤销的命令");
    }
}
public void redoPressed(){
    if(this.redoCmds.size()>0){
        //取出最后一个命令来重做
        Command cmd = this.redoCmds.get(redoCmds.size()-1);
        cmd.execute();
        //把这个命令记录到可撤销的历史记录里面
```

```
            this.undoCmds.add(cmd);
            //然后把最后一个命令删除掉
            this.redoCmds.remove(cmd);
        }else{
            System.out.println("很抱歉，没有可恢复的命令");
        }
    }
}
```

（5）终于到可以收获的时候了，写个客户端，组装好命令和接收者，然后操作几次命令，来测试一下撤销和恢复的功能。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //1：组装命令和接收者
        //创建接收者
        OperationApi operation = new Operation();
        //创建命令对象，并组装命令和接收者
        AddCommand addCmd = new AddCommand(operation,5);
        SubstractCommand substractCmd =
                               new SubstractCommand(operation,3);

        //2：把命令设置到持有者，也就是计算器里面
        Calculator calculator = new Calculator();
        calculator.setAddCmd(addCmd);
        calculator.setSubstractCmd(substractCmd);

        //3:模拟按下按钮，测试一下
        calculator.addPressed();
        System.out.println("一次加法运算后的结果为："
                               +operation.getResult());
        calculator.substractPressed();
        System.out.println("一次减法运算后的结果为："
                               +operation.getResult());

        //测试撤销
        calculator.undoPressed();
        System.out.println("撤销一次后的结果为："
                                 +operation.getResult());
        calculator.undoPressed();
        System.out.println("再撤销一次后的结果为："
                                 +operation.getResult());
```

```
        //测试恢复
        calculator.redoPressed();
        System.out.println("恢复操作一次后的结果为："
                                    +operation.getResult());
        calculator.redoPressed();
        System.out.println("再恢复操作一次后的结果为："
                                      +operation.getResult());
    }
}
```

（6）运行一下，看看结果，享受一下可以撤销和恢复的操作。结果如下：

```
一次加法运算后的结果为：5
一次减法运算后的结果为：2
撤销一次后的结果为：5
再撤销一次后的结果为：0
恢复操作一次后的结果为：5
再恢复操作一次后的结果为：2
```

初始值为 0，执行的两次命令操作为先加上 5，然后再减去 3

13.3.4　宏命令

什么是宏命令呢？简单点说就是包含多个命令的命令，是一个命令的组合。举个例子来说吧，设想一下你去饭店吃饭的过程。

（1）你走进一家饭店，找到座位坐下；

（2）服务员走过来，递给你菜谱；

（3）你开始点菜，服务员开始记录菜单，菜单是三联的，点菜完毕，服务员就会把菜单分成三份，一份给后厨，一份给收银台，一份保留备查；

（4）点完菜，你坐在座位上等候，后厨会按照菜单做菜；

（5）每做好一份菜，就会由服务员送到你桌子上；

（6）然后你就可以大快朵颐了。

事实上，到饭店点餐是一个很典型的命令模式应用。作为客户的你，只需要发出命令，就是要吃什么菜，每道菜就相当于一个命令对象，服务员会在菜单上记录你点的菜，然后把菜单传递给后厨，后厨拿到菜单，会按照菜单进行饭菜制作，后厨就相当于接收者，是命令的真正执行者，厨师才知道每道菜具体怎么实现。

在这个过程中，地位比较特殊的是服务员，在不考虑更复杂的管理，比如后厨管理的时候，负责命令和接收者的组装的就是服务员。比如你点了凉菜、热菜，你其实是不知道到底凉菜由谁来完成，热菜由谁来完成的，因此你只管发命令，而组装的工作就由服务员完成了，服务员知道凉菜送到凉菜部，那是已经做好的了，热菜才送到后厨，需要厨师现做，看起来服务员是一个组装者。

同时呢，服务员还持有命令对象，也就是菜单，最后启动命令执行的也是服务员。

因此，服务员就相当于标准命令模式中的 Client 和 Invoker 的融合。

画个图来描述上述对应关系，如图 13.6 所示。

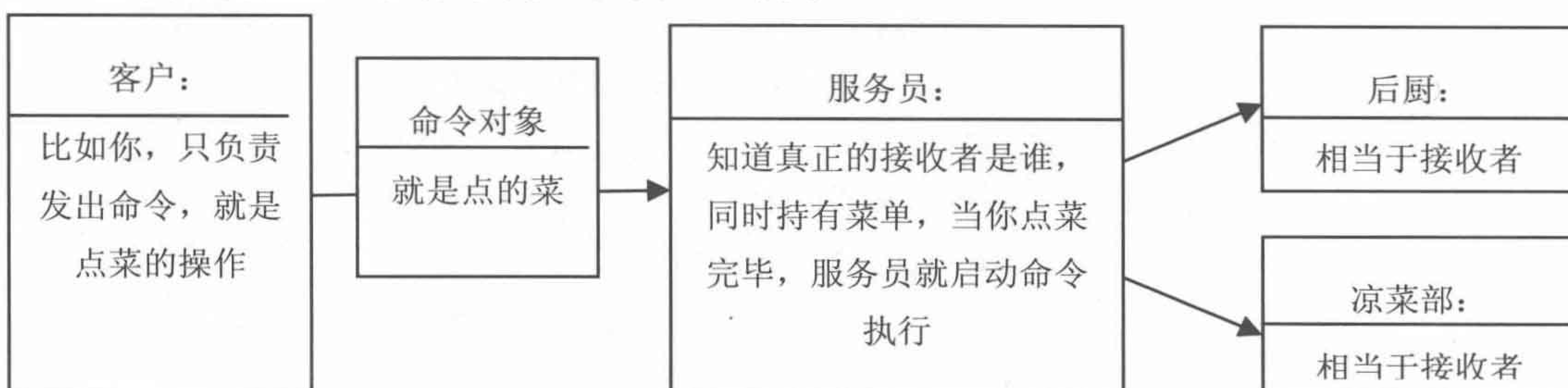


图 13.6　点菜行为与命令模式对应示意图

1. 宏命令在哪里

仔细观察上面的过程，再想想前面的命令模式的实现，看出点什么没有？

前面实现的命令模式都是客户端发出一个命令，然后马上就执行了这个命令，但是在上面的描述里面呢？是点一个菜，服务员就告诉厨师，然后厨师就开始做吗？很明显不是的，服务员会一直等，等到你点完菜，当你说“点完了”的时候，服务员才会启动命令的执行，请注意，这个时候执行的就不是一个命令了，而是执行一堆命令。

描述这一堆命令的就是菜单，如果把菜单也抽象成为一个命令，就相当于一个大的命令，当客户说“点完了”的时候，就相当于触发这个大的命令，意思就是执行菜单这个命令就可以了，这个菜单命令包含多个命令对象，一个命令对象就相当于一道菜。

那么这个**菜单就相当于我们说的宏命令。**

2. 如何实现宏命令

宏命令从本质上讲类似于一个命令，基本上把它当命令对象进行处理。但是它跟普通的命令对象又有些不一样，就是宏命令包含有多个普通的命令对象，简单点说，执行一个宏命令，就是执行宏命令里面所包含的所有命令对象，有点打包执行的意味。

（1）先来定义接收者，就是厨师的接口和实现，先看接口。示例代码如下：

```
/**
 * 厨师的接口
 */
public interface CookApi {
    /**
     * 示意，做菜的方法
     * @param name 菜名
     */
    public void cook(String name);
}
```

厨师又分成两类，一类是做热菜的师傅；一类是做凉菜的师傅，先看看做热菜的厨师的实现示意。示例代码如下：

```
/**
 * 厨师对象，做热菜
 */
public class HotCook implements CookApi{
    public void cook(String name) {
        System.out.println("本厨师正在做："+name);
    }
}
```

做凉菜的师傅示例代码如下：

```
/**
 * 厨师对象，做凉菜
 */
public class CoolCook implements CookApi {
    public void cook(String name) {
        System.out.println("凉菜"+name+"已经做好，本厨师正在装盘。" );
    }
}
```

（2）接下来定义命令接口，和以前一样。示例代码如下：

```
/**
 * 命令接口，声明执行的操作
 */
public interface Command {
    /**
     * 执行命令对应的操作
     */
    public void execute();
}
```

（3）定义好了命令的接口，该来具体实现命令了。

实现方式和以前一样，持有接收者，当执行命令的时候，转调接收者，让接收者去真正实现功能，这里的接收者就是厨师。

> 提示　**这里实现命令的时候，跟标准的命令模式的命令实现有一点不同，标准的命令模式的命令实现的时候，是通过构造方法传入接收者对象，这里改成了使用 setter 的方式来设置接收者对象，也就是说可以动态地切换接收者对象，而无须重新构建对象。**

示例中定义了三道菜，分别是两道热菜：北京烤鸭、绿豆排骨煲，一道凉菜：蒜泥白肉，三个具体的实现类非常类似，只是菜名不同，为了节省篇幅，这里就只看一个命令对象的具体实现。代码示例如下：

```
/**
 * 命令对象，绿豆排骨煲
 */
public class ChopCommand implements Command{
    /**
     * 持有具体做菜的厨师的对象
     */
    private CookApi cookApi = null;
    /**
     * 设置具体做菜的厨师的对象
     * @param cookApi 具体做菜的厨师的对象
     */
    public void setCookApi(CookApi cookApi) {
        this.cookApi = cookApi;
    }

    public void execute() {
        this.cookApi.cook("绿豆排骨煲");
    }
}
```


（4）该来组合菜单对象了，也就是宏命令对象。

① 宏命令就其本质还是一个命令，所以一样要实现 Command 接口。

② 宏命令和普通命令的不同在于：宏命令是多个命令组合起来的，因此在宏命令对象里面会记录多个组成它的命令对象。

③ 既然是包含多个命令对象，得有方法让这么多个命令对象能被组合进来。

④ 既然宏命令包含了多个命令对象，执行宏命令对象就相当于依次执行这些命令对象，也就是循环执行这些命令对象

看看代码示例会更清晰些。代码示例如下：

```
/**
 * 菜单对象，是个宏命令对象
 */
public class MenuCommand implements Command {
    /**
     * 用来记录组合本菜单的多道菜品，也就是多个命令对象
     */
    private Collection<Command> col = new ArrayList<Command>();
    /**
     * 点菜，把菜品加入到菜单中
     * @param cmd 客户点的菜
```

```
     */
    public void addCommand(Command cmd){
        col.add(cmd);
    }
    public void execute() {
        //执行菜单其实就是循环执行菜单里面的每个菜
        for(Command cmd : col){
            cmd.execute();
        }
    }
}
```

（5）该服务员类重磅登场了，它实现的功能，相当于标准命令模式实现中的 Client 加上 Invoker，前面都是文字讲述，看看代码如何实现。示例代码如下：

```
/**
 * 服务员，负责组合菜单，负责组装每个菜和具体的实现者
 * 还负责执行调用，相当于标准Command模式的Client+Invoker
 */
public class Waiter {
    /**
     * 持有一个宏命令对象—菜单
     */
    private MenuCommand menuCommand = new MenuCommand();
    /**
     * 客户点菜
     * @param cmd 客户点的菜，每道菜是一个命令对象
     */
    public void orderDish(Command cmd){
        //客户传过来的命令对象是没有和接收者组装的
        //在这里组装吧
        CookApi hotCook = new HotCook();
        CookApi coolCook = new CoolCook();
        //判读到底是组合凉菜师傅还是热菜师傅
        //简单点根据命令的原始对象的类型来判断
        if(cmd instanceof DuckCommand){
            ((DuckCommand)cmd).setCookApi(hotCook);
        }else if(cmd instanceof ChopCommand){
            ((ChopCommand)cmd).setCookApi(hotCook);
        }else if(cmd instanceof PorkCommand){
            //这是个凉菜，所以要组合凉菜的师傅
            ((PorkCommand)cmd).setCookApi(coolCook);
```

```
        }
        //添加到菜单中
        menuCommand.addCommand(cmd);
    }
    /**
     * 客户点菜完毕，表示要执行命令了，这里就是执行菜单这个组合命令
     */
    public void orderOver(){
        this.menuCommand.execute();
    }
}
```

（6）费了这么大力气，终于可以坐下来歇息一下，点菜吃饭吧，一起来看看客户端怎样使用这个宏命令，其实在客户端非常简单，根本看不出宏命令来，代码示例如下：

```
public class Client {
    public static void main(String[] args) {
        //客户只是负责向服务员点菜就好了
        //创建服务员
        Waiter waiter = new Waiter();

        //创建命令对象，就是要点的菜
        Command chop = new ChopCommand();
        Command duck = new DuckCommand();
        Command pork = new PorkCommand();

        //点菜，就是把这些菜让服务员记录下来
        waiter.orderDish(chop);
        waiter.orderDish(duck);
        waiter.orderDish(pork);

        //点菜完毕
        waiter.orderOver();
    }
}
```

运行一下，享受一下成果。结果如下：

```
本厨师正在做：绿豆排骨煲
本厨师正在做：北京烤鸭
凉菜蒜泥白肉已经做好，本厨师正在装盘。
```

13.3.5　队列请求

所谓队列请求，就是对命令对象进行排队，组成工作队列，然后依次取出命令对象来执行。通常用多线程或者线程池来进行命令队列的处理，当然也可以不用多线程，就是一个线程，一个命令一个命令地循环处理，就是慢了点。

继续宏命令的例子。其实在后厨，会收到很多菜单，一般是按照菜单传递到后厨的先后顺序来进行处理。对每张菜单，假定也是按照菜品的先后顺序进行制作，那么在后厨则自然形成了一个菜品的队列，也就是很多个用户的命令对象的队列。

后厨有很多厨师，每个厨师都从这个命令队列里面取出一个命令，然后按照命令做出菜来，就相当于多个线程在同时处理一个队列请求。

因此后厨就是一个典型的队列请求的例子。

> 注意　**后厨的厨师与命令队列之间是没有任何关联的，也就是说是完全解耦的。命令队列是客户发出的命令，厨师只是负责从队列里面取出一个，处理，然后再取下一个，再处理，仅此而已，厨师不知道也不管客户是谁。**

下面来看看如何实现队列请求。

1. 如何实现命令模式的队列请求

（1）先从命令接口开始。除了 execute 方法外，新增加了一个返回发出命令的桌号，就是点菜的桌号，还有一个是为命令对象设置接收者的方法，也把它添加到接口上，这个是为了后面多线程处理的时候方便使用。示例代码如下：

```
/**
 * 命令接口，声明执行的操作
 */
public interface Command {
    /**
     * 执行命令对应的操作
     */
    public void execute();
    /**
     * 设置命令的接收者
     * @param cookApi 命令的接收者
     */
    public void setCookApi(CookApi cookApi);
    /**
     * 返回发起请求的桌号，就是点菜的桌号
     * @return 发起请求的桌号
     */
```

```
    public int getTableNum();
}
```

（2）厨师的接口也发生了一点变化，在 cook 的方法上添加了发出命令的桌号。这样，在多线程输出信息的时候，才知道到底是在给哪个桌做菜。示例代码如下：

```
/**
 * 厨师的接口
 */
public interface CookApi {
    /**
     * 示意，做菜的方法
     * @param tableNum 点菜的桌号
     * @param name 菜名
     */
    public void cook(int tableNum,String name);
}
```

（3）开始来实现命令接口。为了简单，这次只有热菜，因为要做的工作都在后厨的命令队列里面，因此凉菜就不要了。示例代码如下：

```
/**
 * 命令对象，绿豆排骨煲
 */
public class ChopCommand implements Command{
    /**
     * 持有具体做菜的厨师的对象
     */
    private CookApi cookApi = null;
    /**
     * 设置具体做菜的厨师的对象
     * @param cookApi 具体做菜的厨师的对象
     */
    public void setCookApi(CookApi cookApi) {
        this.cookApi = cookApi;
    }
    /**
     * 点菜的桌号
     */
    private int tableNum;
    /**
     * 构造方法，传入点菜的桌号
     * @param tableNum 点菜的桌号
```

```
     */
    public ChopCommand(int tableNum){
        this.tableNum = tableNum;
    }
    public int getTableNum(){
        return this.tableNum;
    }
    public void execute() {
        this.cookApi.cook(tableNum,"绿豆排骨煲");
    }
}
```

还有一个命令对象是“北京烤鸭”，和上面的实现一样，只是菜名不同而已，所以就不去展示示例代码了。

（4）接下来构建很重要的命令对象的队列。其实也不是有多难，多个命令对象可以用一个集合来存储就可以了，然后按照放入的顺序，先进先出即可。

请注意：为了演示的简单性，这里没有使用java.util.Queue，直接使用List来模拟实现了。

示例代码如下：

```
/**
 * 命令队列类
 */
public class CommandQueue {
    /**
* 用来存储命令对象的队列
*/
    private static List<Command> cmds = new ArrayList<Command>();
    /**
     * 服务员传过来一个新的菜单，需要同步
     * 因为同时会有很多的服务员传入菜单，而同时又有很多厨师在从队列里取值
     * @param menu 传入的菜单
     */
    public synchronized static void addMenu(MenuCommand menu){
        //一个菜单对象包含很多命令对象
        for(Command cmd : menu.getCommands()){
            cmds.add(cmd);
        }
    }
    /**
     * 厨师从命令队列里面获取命令对象进行处理，也是需要同步的
```

```
    */
    public synchronized static Command getOneCommand(){
        Command cmd = null;
        if(cmds.size() > 0 ){
            //取出队列的第一个，因为是约定的按照加入的先后来处理
            cmd = cmds.get(0);
            //同时从队列里面取掉这个命令对象
            cmds.remove(0);
        }
        return cmd;
    }
}
```

> **注意** 这里并没有考虑一些复杂的情况，比如，如果命令队列里面没有命令，而厨师又来获取命令该怎么办？这里只是做了一个基本的示范，并没有完整的实现，所以也就没有去处理这些问题。当然，出现这种问题，需要使用 wait/notify 来进行线程调度。

（5）有了命令队列，谁来向这个队列里面传入命令呢？

很明显是服务员，当客户点菜完成，服务员就会执行菜单，现在执行菜单就相当于把菜单直接传递给后厨，也就是要把菜单里的所有命令对象加入到命令队列里面，因此菜单对象的实现需要改变。示例代码如下：

```
/**
 * 菜单对象，是个宏命令对象
 */
public class MenuCommand implements Command {
    /**
      * 用来记录组合本菜单的多道菜品，也就是多个命令对象
*/
    private Collection<Command> col = new ArrayList<Command>();
    /**
     * 点菜，把菜品加入到菜单中
     * @param cmd 客户点的菜
     */
    public void addCommand(Command cmd){
        col.add(cmd);
    }
```

```
    public void setCookApi(CookApi cookApi){
        //什么都不用做
    }
    public int getTableNum(){
        //什么都不用做
        return 0;
    }
    /**
     * 获取菜单中的多个命令对象
     * @return 菜单中的多个命令对象
     */
    public Collection<Command> getCommands(){
        return this.col;
    }

    public void execute() {
        //执行菜单就是把菜单传递给后厨
        CommandQueue.addMenu(this);
    }
}
```

这两个方法对组合命令对象的菜单没有实际意义

这里发生了改变，以前是循环执行每个命令

（6）现在有了命令队列，也有人负责向队列里面添加命令了，可是谁来执行命令队列里面的命令呢？

答案是：由厨师从命令队列里面获取命令，并真正处理命令，而且厨师在处理命令前会把自己设置到命令对象里面去当接收者，表示这个菜由我来实际做。

厨师对象的实现大致有以下的改变。

- 为了更好地体现命令队列的用法，而实际情况也是多个厨师，这里用多线程来模拟多个厨师。他们自己从命令队列里面获取命令，然后处理命令，再获取下一个，如此反复。因此厨师类要实现多线程接口。
- 还有一个改变，为了在多线程中输出信息，让我们知道是哪一个厨师在执行命令，给厨师添加了一个姓名的属性，通过构造方法传入。
- 另外一个改变是为了在多线程中看出效果，在厨师真正做菜的方法里面使用随机数模拟了一个做菜的时间。

好了，介绍完了改变的地方，一起看看代码吧。示例代码如下：

```
/**
 * 厨师对象，做热菜的厨师
 */
public class HotCook implements CookApi,Runnable{
    /**
     * 厨师姓名
```

```
     */
    private String name;
    /**
     * 构造方法，传入厨师姓名
     * @param name 厨师姓名
     */
    public HotCook(String name){
        this.name = name;
    }
    public void cook(int tableNum,String name) {
        //每次做菜的时间是不一定的，用随机数来模拟一下
        int cookTime = (int)(20 * Math.random());
        System.out.println(this.name+"厨师正在为"+tableNum
                                        +"号桌做："+name);
        try {
            //让线程休息这么长时间，表示正在做菜
            Thread.sleep(cookTime);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(this.name+"厨师为"+tableNum
                +"号桌做好了："+name+"，共计耗时="+cookTime+"秒");
    }
    public void run() {
        while(true){
            //从命令队列里面获取命令对象
            Command cmd = CommandQueue.getOneCommand();
            if(cmd != null){
                //说明取到命令对象了，这个命令对象还没有设置接收者
                //因为前面还不知道到底哪一个厨师来真正执行这个命令
                //现在知道了，就是当前厨师实例，设置到命令对象里面
                cmd.setCookApi(this);
                //然后真正执行这个命令
                cmd.execute();
            }
            //休息1秒
            try {
                Thread.sleep(1000L);
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }
    }
}
```

（7）下面该来看看服务员类了。由于现在考虑了后厨的管理，因此从实际情况来看，这次服务员也不知道到底命令的真正接收者是谁了，也就是说服务员也不知道某个菜最后到底由哪一位厨师完成，所以服务员类就简单了。

组装命令对象和接收者的功能后移到厨师类的线程里面了。当某个厨师从命令队列里面获取一个命令对象的时候，这个厨师就是这个命令的真正接收者。

服务员类的示例代码如下：

```
/**
 * 服务员，负责组合菜单，还负责执行调用
 */
public class Waiter {
    /**
     * 持有一个宏命令对象——菜单
     */
    private MenuCommand menuCommand = new MenuCommand();
    /**
     * 客户点菜
     * @param cmd 客户点的菜，每道菜是一个命令对象
     */
    public void orderDish(Command cmd){
        //添加到菜单中
        menuCommand.addCommand(cmd);
    }
    /**
     * 客户点菜完毕，表示要执行命令了，这里就是执行菜单这个组合命令
     */
    public void orderOver(){
        this.menuCommand.execute();
    }
}
```

（8）在见到曙光之前，还有一个问题要解决，就是谁来启动多线程的厨师呢？

为了实现后厨的管理，为此专门定义一个后厨管理的类，在这个类里面启动多个厨师的线程，而且这种启动在运行期间只有一次。示例代码如下：

```
/**
 * 后厨的管理类，通过此类让后厨的厨师进行运行状态
 */
```

```
public class CookManager {
    /**
     * 用来控制是否需要创建厨师，如果已经创建了就不要再执行了
     */
    private static boolean runFlag = false;
    /**
     * 运行厨师管理，创建厨师对象并启动他们相应的线程
     * 无论运行多少次，创建厨师对象和启动线程的工作只做一次
     */
    public static void runCookManager(){
        if(!runFlag){
            runFlag = true;
            //创建三位厨师
            HotCook cook1 = new HotCook("张三");
            HotCook cook2 = new HotCook("李四");
            HotCook cook3 = new HotCook("王五");

            //启动他们的线程
            Thread t1 = new Thread(cook1);
            t1.start();
            Thread t2 = new Thread(cook2);
            t2.start();
            Thread t3 = new Thread(cook3);
            t3.start();
        }
    }
}
```

（9）下面写个客户端测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //先要启动后台，让整个程序运行起来
        CookManager.runCookManager();

        //为了简单，直接用循环模拟多个桌号点菜
        for(int i = 0;i<5;i++){
            //创建服务员
            Waiter waiter = new Waiter();
            //创建命令对象，就是要点的菜
            Command chop = new ChopCommand(i);
            Command duck = new DuckCommand(i);
```

```
            //点菜，就是把这些菜让服务员记录下来
            waiter.orderDish(chop);
            waiter.orderDish(duck);

            //点菜完毕
            waiter.orderOver();
        }
    }
}
```

（10）运行一下，看看效果。因为是使用多线程在处理请求队列，可能每次运行的效果不一样。某次运行的结果如下：

```
张三厨师正在为0号桌做：绿豆排骨煲
张三厨师为0号桌做好了：绿豆排骨煲，共计耗时=13秒
王五厨师正在为0号桌做：北京烤鸭
李四厨师正在为1号桌做：绿豆排骨煲
李四厨师为1号桌做好了：绿豆排骨煲，共计耗时=5秒
王五厨师为0号桌做好了：北京烤鸭，共计耗时=18秒
张三厨师正在为1号桌做：北京烤鸭
张三厨师为1号桌做好了：北京烤鸭，共计耗时=1秒
李四厨师正在为2号桌做：绿豆排骨煲
李四厨师为2号桌做好了：绿豆排骨煲，共计耗时=12秒
王五厨师正在为2号桌做：北京烤鸭
王五厨师为2号桌做好了：北京烤鸭，共计耗时=7秒
张三厨师正在为3号桌做：绿豆排骨煲
张三厨师为3号桌做好了：绿豆排骨煲，共计耗时=15秒
李四厨师正在为3号桌做：北京烤鸭
王五厨师正在为4号桌做：绿豆排骨煲
李四厨师为3号桌做好了：北京烤鸭，共计耗时=17秒
王五厨师为4号桌做好了：绿豆排骨煲，共计耗时=16秒
张三厨师正在为4号桌做：北京烤鸭
张三厨师为4号桌做好了：北京烤鸭，共计耗时=0秒
```


仔细观察上面的数据。在多线程环境下，虽然保障了命令对象取出的顺序是先进先出，但是究竟是哪一个厨师来做，还有具体做多长时间都是不定的。

13.3.6　日志请求

所谓日志请求，就是把请求的历史记录保存下来，一般是采用永久存储的方式。如果在运行请求的过程中，系统崩溃了，那么当系统再次运行时，就可以从保存的历史记

录中获取日志请求，并重新执行命令。

日志请求的实现有两种方案：一种是直接使用 Java 中的序列化方法；另外一种就是在命令对象中添加上存储和装载的方法，其实就是让命令对象自己实现类似序列化的功能。当然要简单就直接使用 Java 中的序列化。

结合前面队列请求的例子，来简单演示一下日志请求的功能。

考虑在队列请求的例子里面，当菜单都被传到后台以后，后台会把这些菜单做成一个请求队列，然后让厨师从这个队列里面获取命令去执行。这里就存在一个问题，如果这个系统运行中突然崩溃了呢？比如突然断电了，那该怎么办呢？

难道等系统再次运行的时候，后厨要求服务员再把菜单全部重新传递一次吗？即使服务员全部传一次，那样也还是有问题，因为有些命令是已经被执行了的，它们不应该被重复执行，而服务员是不知道哪些命令是已经执行了的。

> 提示 **一个可行的解决方案就是把这个请求队列日志化，当有新的菜单传递过来的时候，更新日志，当厨师从队列里面获取一个请求去执行的时候，这个请求就应该从日志中去掉，这样即使系统崩溃了，也能够恢复，而且恢复的都是还没有执行的命令。**

1. 实现日志请求

由于篇幅关系，就不去做比较复杂的示例了，直接沿用前面对列请求的例子，采用 Java 中序列化的方法，这样最简单。

（1）先序列化命令对象，因为要把这些对象保存到文件中去。在 ChopCommand 和 DuckCommand 对象上实现 java.io.Serializable，这个就不用代码示例了。

（2）前面的实现中把请求队列实现成了 List 对象，为了保存这个 List 对象，设计一个文件操作的工具类，来实现向文件中写入 List 和从文件中获取 List 对象。示例代码如下：

```
/**
 * 读写文件的辅助工具类
 */
public class FileOpeUtil {
    /**
     * 私有化构造方法，避免外部无谓的创建类实例
     * 这个工具类不需要创建类实例
     */
    private FileOpeUtil() {
    }
    /**
     * 读文件，从文件中获取存储的List对象
     * @param pathName 文件路径和文件名
```

```
     * @return 存储的List对象
     */
    public static List readFile(String pathName) {
        List list = new ArrayList();
        ObjectInputStream oin = null;
        try {
            File f = new File(pathName);
            if(f.exists()){
                oin = new ObjectInputStream(
                        new BufferedInputStream(
                        new FileInputStream(f)));
                list = (List)oin.readObject();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            try {
                if(oin!=null){
                    oin.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return list;
    }
    /**
     * 写文件，把List对象写到文件中
     * @param pathName 文件路径和文件名
     * @param list 要写到文件的List对象
     */
    public static void writeFile(String pathName,List list){
        File f = new File(pathName);
        ObjectOutputStream oout = null;
        try {
            oout = new ObjectOutputStream(
                    new BufferedOutputStream(
                    new FileOutputStream(f)));
            oout.writeObject(list);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }finally{
            try {
                oout.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

（3）有了读写文件的工具类，下面来看看在命令队列里面如何实现把队列日志化了。其实思路很简单，就是在装载队列的时候，先从日志中获取上一次还没有做完的命令。如果没有，那就新建一个队列，然后在每次向队列里面添加命令的时候就更新日志；如果有厨师取出命令，那就删除掉该命令，并重新更新日志。示例代码如下：

```
public class CommandQueue {
    /**
     * 新添加的，文件名称
     */
    private final static String FILE_NAME = "CmdQueue.txt";
    /**
* 用来存储命令对象的队列
*/
    private static List<Command> cmds = null;

    static{
        //获取上次没有做完的命令队列
        cmds = FileOpeUtil.readFile(FILE_NAME);
        if(cmds==null){
            cmds = new ArrayList<Command>();
        }
    }

    /**
* 服务员传过来一个新的菜单，需要同步
*/
    public synchronized static void addMenu(MenuCommand menu){
        //一个菜单对象包含很多命令对象
        for(Command cmd : menu.getCommands()){
            cmds.add(cmd);
        }
```

从日志中获取上一次还没有做完的命令，如果没有，那就新建一个队列

```
        //记录请求日志
        FileOpeUtil.writeFile(FILE_NAME, cmds);
    }
    /**
* 厨师从命令队列里面获取命令对象进行处理，也是需要同步的
*/
    public synchronized static Command getOneCommand(){
        Command cmd = null;
        if(cmds.size() > 0 ){
            //取出队列的第一个，因为约定的是按照加入的先后来处理
            cmd = cmds.get(0);
            //同时从队列里面删除这个命令对象
            cmds.remove(0);
            //记录请求日志
            FileOpeUtil.writeFile(FILE_NAME, cmds);
        }
        return cmd;
    }
}
```

（4）其他部分和队列请求的例子完全一样，就不再赘述。

可以运行测试代码来看看效果，要想看出在中断后，下次运行时是否能够恢复上次没有运行的命令，可以在第一次运行的时候，也就是多线程还没有处理完全部命令的时候强制终止运行，然后再启动看看，是否能够恢复上次没有执行完的命令。

第一次运行客户端，中途强制终止。示例的结果如下：

```
张三厨师正在为0号桌做：绿豆排骨煲
李四厨师正在为0号桌做：北京烤鸭
李四厨师为0号桌做好了：北京烤鸭，共计耗时=2秒
王五厨师正在为1号桌做：绿豆排骨煲
张三厨师为0号桌做好了：绿豆排骨煲，共计耗时=15秒
王五厨师为1号桌做好了：绿豆排骨煲，共计耗时=7秒
李四厨师正在为1号桌做：北京烤鸭
李四厨师为1号桌做好了：北京烤鸭，共计耗时=3秒
王五厨师正在为2号桌做：绿豆排骨煲
张三厨师正在为2号桌做：北京烤鸭
王五厨师为2号桌做好了：绿豆排骨煲，共计耗时=14秒
张三厨师为2号桌做好了：北京烤鸭，共计耗时=5秒
```

注意第一次运行终止的时候，刚好把第三桌的菜做好。下面再次运行客户端，应该先要把刚才没有做完的菜做完，然后才继续新的菜单。示例的结果如下：

```
张三厨师正在为3号桌做：绿豆排骨煲
王五厨师正在为3号桌做：北京烤鸭
李四厨师正在为4号桌做：绿豆排骨煲
王五厨师为3号桌做好了：北京烤鸭，共计耗时=3秒
张三厨师为3号桌做好了：绿豆排骨煲，共计耗时=9秒
李四厨师为4号桌做好了：绿豆排骨煲，共计耗时=19秒
王五厨师正在为4号桌做：北京烤鸭
张三厨师正在为0号桌做：绿豆排骨煲
张三厨师为0号桌做好了：绿豆排骨煲，共计耗时=0秒
王五厨师为4号桌做好了：北京烤鸭，共计耗时=13秒
李四厨师正在为0号桌做：北京烤鸭
李四厨师为0号桌做好了：北京烤鸭，共计耗时=17秒
张三厨师正在为1号桌做：绿豆排骨煲
张三厨师为1号桌做好了：绿豆排骨煲，共计耗时=2秒
王五厨师正在为1号桌做：北京烤鸭
...
```

注意这一段，是在完成系统崩溃时还没有执行的命令，是从日志请求里面恢复的

执行完上次的命令，继续新的命令，为节省篇幅，后面就省略了

2. 小结

通过上面的示例可以看出，实现日志请求也不是很麻烦，当然，如果需要自己做序列化会复杂一些。对于扩展日志请求，在高级应用中，可以扩展到事务的处理中，因为事务的基本实现机制就是先写日志，然后再操作数据库，这里就不再继续展开了。

13.3.7 命令模式的优点

- 更松散的耦合

  命令模式使得发起命令的对象——客户端，和具体实现命令的对象——接收者对象完全解耦，也就是说发起命令的对象完全不知道具体实现对象是谁，也不知道如何实现。

- 更动态的控制

  命令模式把请求封装起来，可以动态地对它进行参数化、队列化和日志化等操作，从而使得系统更灵活。

- 很自然的复合命令

  命令模式中的命令对象能够很容易地组合成复合命令，也就是前面讲的宏命令，从而使系统操作更简单，功能更强大。

- 更好的扩展性

  由于发起命令的对象和具体的实现完全解耦，因此扩展新的命令就很容易，只需要实现新的命令对象，然后在装配的时候，把具体的实现对象设置到命令对象中，然后就可以使用这个命令对象，已有的实现完全不用变化。

13.3.8　思考命令模式

1. 命令模式的本质

> 命令模式的本质：封装请求。

前面讲了，命令模式的关键就是把请求封装成为命令对象，然后就可以对这个对象进行一系列的处理了，比如上面讲到的参数化配置、可撤销操作、宏命令、队列请求、日志请求等功能处理。

2. 何时选用命令模式

建议在以下情况时选用命令模式。

- 如果需要抽象出需要执行的动作，并参数化这些对象，可以选用命令模式。将这些需要执行的动作抽象成为命令，然后实现命令的参数化配置。
- 如果需要在不同的时刻指定、排列和执行请求，可以选用命令模式。将这些请求封装成为命令对象，然后实现将请求队列化。
- 如果需要支持取消操作，可以选用命令模式，通过管理命令对象，能很容易地实现命令的恢复和重做功能。
- 如果需要支持当系统崩溃时，能将系统的操作功能重新执行一遍，可以选用命令模式。将这些操作功能的请求封装成命令对象，然后实现日志命令，就可以在系统恢复以后，通过日志获取命令列表，从而重新执行一遍功能。
- 在需要事务的系统中，可以选用命令模式。命令模式提供了对事务进行建模的方法。命令模式有一个别名就是 Transaction。

13.3.9　退化的命令模式

在领会了命令模式的本质后，接下来思考一个命令模式退化的情况。

前面讲到了智能命令，如果命令的实现对象超级智能，实现了命令要求的所有功能，那么就不需要接收者了，既然没有了接收者，那么也就不需要组装者了。

（1）举个最简单的示例来说明。

比如现在要实现一个打印服务，由于非常简单，所以基本上就没有什么讲述，依次来看，命令接口定义如下：

```
public interface Command {
    public void execute();
}
```

命令的实现示例代码如下：

```
public class PrintService implements Command{
    /**
```

```
     * 要输出的内容
     */
    private String str = "";
    /**
     * 构造方法，传入要输出的内容
     * @param s 要输出的内容
     */
    public PrintService(String s){
        str = s;
    }
    public void execute() {
        System.out.println("打印的内容为="+str);
    }
}
```

智能的体现，自己知道怎么实现命令所要求的功能，并真正地实现了相应的功能，不再转调接收者了

此时的 Invoker 示例代码如下：

```
public class Invoker {
    /**
     * 持有命令对象
     */
    private Command cmd = null;
    /**
     * 设置命令对象
     * @param cmd 命令对象
     */
    public void setCmd(Command cmd){
        this.cmd = cmd;
    }
    /**
     * 开始打印
     */
    public void startPrint(){
        //执行命令的功能
        this.cmd.execute();
    }
}
```

最后看看客户端的代码。示例如下：

```
public class Client {
    public static void main(String[] args) {
        //准备要发出的命令
```

```
        Command cmd = new PrintService("退化的命令模式示例");
        //设置命令给持有者
        Invoker invoker = new Invoker();
        invoker.setCmd(cmd);

        //按下按钮，真正启动执行命令
        invoker.startPrint();
    }
}
```

测试结果如下：

```
打印的内容为=退化的命令模式示例
```

（2）继续变化。

如果此时继续变化，Invoker 也开始变得智能化，在 Invoker 的 startPrint 方法里面，Invoker 加入了一些实现，同时 Invoker 对持有命令也有意见，觉得自己是个傀儡，要求改变一下，直接在调用方法的时候传递命令对象进来。示例代码如下：

```
public class Invoker {
    public void startPrint(Command cmd){
        System.out.println("在Invoker中，输出服务前");
        cmd.execute();
        System.out.println("输出服务结束");
    }
}
```


看起来 Invoker 退化成一个方法了。

这个时候 Invoker 很高兴，宣称自己是一个智能的服务，不再是一个傻傻的转调者，而是有自己功能的服务了。这个时候 Invoker 调用命令对象的执行方法，也不叫转调，改名叫“回调”，意思是在我 Invoker 需要的时候，会回调你命令对象，命令对象你就乖乖地写好实现，等我“回调”你就可以了。

事实上这个时候的命令模式的实现基本上就等同于 Java 回调机制的实现。可能有些朋友看起来感觉还不是很像，那是因为在 Java 回调机制的常见实现上，经常没有单独的接口实现类，而是采用匿名内部类的方式来实现的。

（3）再进一步。

把单独实现命令接口的类改成用匿名内部类实现，这个时候就只剩下命令的接口、Invoker 类，还有客户端了。

为了使用匿名内部类，还需要设置输出的值，对命令接口做点小改动，增加一个设置输出值的方法。示例代码如下：

```
public interface Command {
    public void execute();
    /**
     * 设置要输出的内容
     * @param s 要输出的内容
     */
    public void setStr(String s);
}
```

此时 Invoker 就是上面那个，而客户端会有些改变。客户端的示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备要发出的命令，没有具体实现类了
        Command cmd = new Command(){
            private String str = "";
            public void setStr(String s){
                str = s;
            }
            public void execute() {
                System.out.println("打印的内容为="+str);
            }
        };
        cmd.setStr("退化的命令模式类似于Java回调的示例");
        //这个时候的Invoker或许该称为服务了
        Invoker invoker = new Invoker();
        //按下按钮，真正启动执行命令
        invoker.startPrint(cmd);
    }
}
```

匿名内部类来实现命令

运行测试一下。结果如下：

```
在Invoker中，输出服务前
打印的内容为=退化的命令模式类似于Java回调的示例
输出服务结束
```

（4）现在是不是看出来了，这个时候的命令模式的实现基本上就等同于 Java 回调机制的实现。这也是很多人常说的命令模式可以实现 Java 回调的意思。

当然更狠的是连 Invoker 也不要了，直接把那个方法搬到 Client 中，那样测试起来就更方便了。在实际开发中，应用命令模式来实现回调机制的时候，Invoker 通常还是有的，但可以智能化实现，更准确地说 Invoker 充当客户调用的服务实现，而回调的方法只是实现服务功能中的一个或者几个步骤。

13.3.10　相关模式

- 命令模式和组合模式

  这两个模式可以组合使用。

  在命令模式中，实现宏命令的功能就可以使用组合模式来实现。前面的示例并没有按照组合模式来做，那是为了保持示例的简单，还有突出命令模式的实现，这点请注意。

- 命令模式和备忘录模式

  这两个模式可以组合使用。

  在命令模式中，实现可撤销操作功能时，前面讲了有两种实现方式，其中有一种就是保存命令执行前的状态，撤销的时候就把状态恢复。如果采用这种方式实现，就可以考虑使用备忘录模式。

  如果状态存储在命令对象中，那么还可以使用原型模式，把命令对象当作原型来克隆一个新的对象，然后将克隆出来的对象通过备忘录模式存放。

- 命令模式和模板方法模式

  这两个模式从某种意义上有相似的功能，命令模式可以作为模板方法的一种替代模式，也就是说命令模式可以模仿实现模板方法模式的功能。

  如同前面讲述的退化的命令模式可以实现 Java 的回调，而 Invoker 智能化后向服务进化，如果 Invoker 的方法就是一个算法骨架，其中有两步在这个骨架里面没有具体实现，需要外部来实现，这个时候就可以通过回调命令接口来实现。

  而类似的功能在模板方法中，是先调用抽象方法，然后等待子类来实现。

  可以看出虽然实现方式不一样，但是可以实现相同的功能。

读书笔记

第 14 章　迭代器模式（Iterator）

14.1 场景问题

14.1.1 工资表数据的整合

考虑这样一个实际应用：整合工资表数据。

这个项目的背景是这样的，项目的客户方收购了一家小公司，这家小公司有自己的工资系统，现在需要整合到客户方已有的工资系统中。

客户方已有的工资系统，在内部是采用 List 来记录工资列表；而新收购的这家公司的工资系统，在内部是采用数组来记录工资列表。但是幸运的是，两个系统用来描述工资的数据模型是差不多的。

要整合这两个工资系统的工资数据，当然最简单的方式是考虑直接把新收购的这家公司的工资系统也改成内部使用 List 来记录工资列表，但是经过仔细查看源代码，发现有很多的代码跟这个数组相关，还有很多是比较重要的逻辑处理，比如计算工资等，因此只好作罢。

现在除了要把两个工资系统整合起来外，老板还希望能够通过决策辅助系统来统一查看工资数据，他不想看到两份不同的工资表。那么应该如何实现呢？

14.1.2 有何问题

本来就算内部描述形式不一样，只要不需要整合在一起，两个系统单独输出自己的工资表也是没有什么问题的。但是，老板还是希望能够以一个统一的方式来查看所有的工资数据，也就是说从外部看起来，两个系统输出的工资表应该是一样的。

经过分析，既要满足老板的要求，又要让两边的系统改动都尽可能小的话，问题的核心就在于如何能够以一种统一的方式来提供工资数据给决策辅助系统，换句话来说就是：**如何能够以一个统一的方式来访问内部实现不同的聚合对象。**

14.2 解决方案

14.2.1 使用迭代器模式来解决问题

用来解决上述问题的一个合理的解决方案就是迭代器模式。那么什么是迭代器模式呢？

1. 迭代器模式的定义

> **提供一种方法顺序访问一个聚合对象中的各个元素，而又不需暴露该对象的内部表示。**

所谓聚合是指一组对象的组合结构，比如：Java 中的集合、数组等。

2. 是应用迭代器模式来解决问题的思路

仔细分析上面的问题，要以一个统一的方式来访问内部实现不同的聚合对象，那么首先需要把这个统一的访问方式定义出来，按照这个统一的访问方式定义出来的接口，在迭代器模式中对应的就是 Iterator 接口。

迭代器迭代的是具体的聚合对象，那么不同的聚合对象就应该有不同的迭代器，为了让迭代器以一个统一的方式来操作聚合对象，因此给所有的聚合对象抽象出一个公共的父类，让它提供操作聚合对象的公共接口，这个抽象的公共父类在迭代器模式中对应的就是 Aggregate 对象。

接下来就该考虑如何创建迭代器了。由于迭代器和相应的聚合对象紧密相关，因此让具体的聚合对象来负责创建相应的迭代器对象。

14.2.2　迭代器模式的结构和说明

迭代器模式的结构如图 14.1 所示。

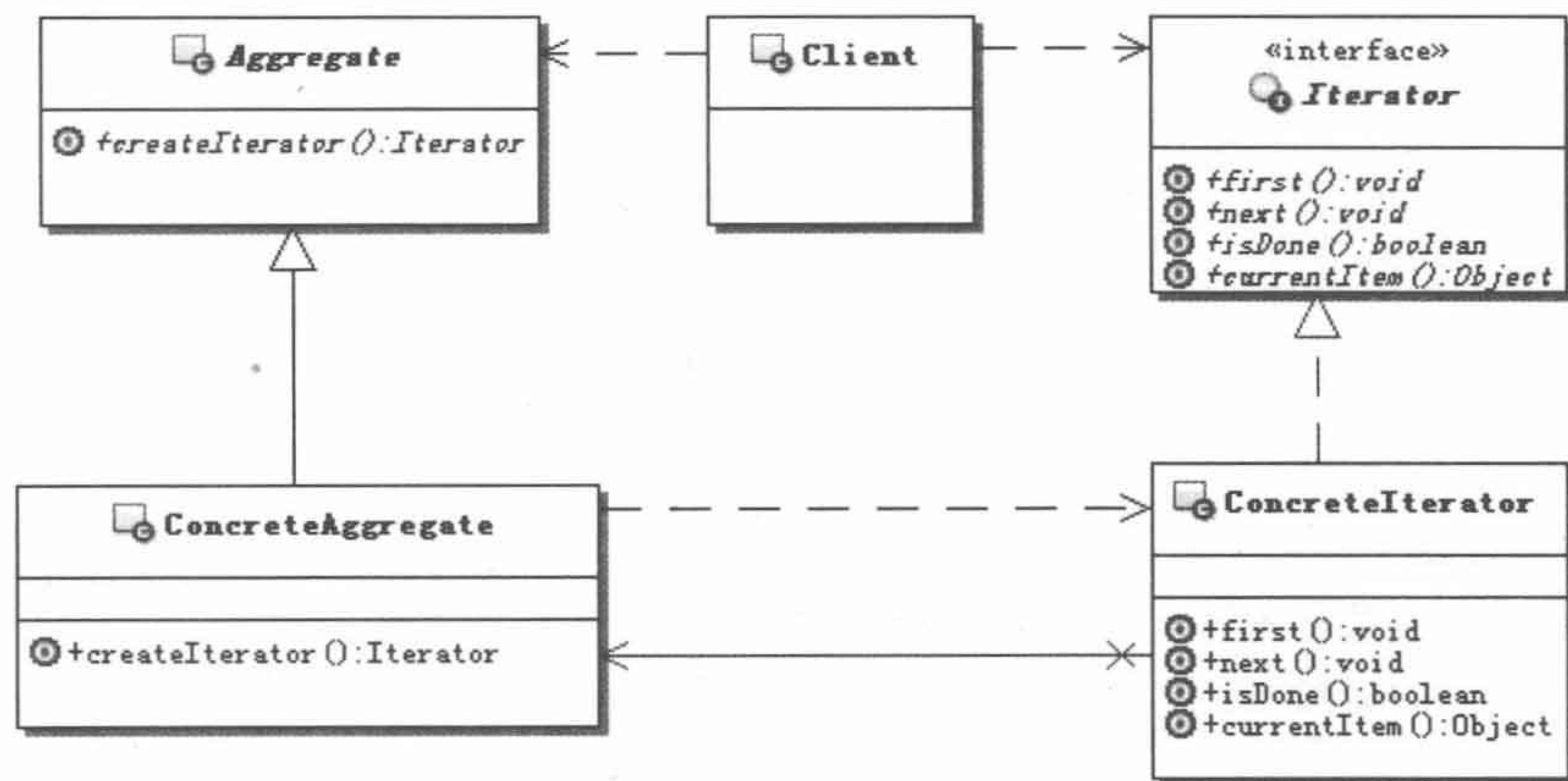


图 14.1　迭代器模式的结构示意图

- Iterator：迭代器接口。定义访问和遍历元素的接口。
- ConcreteIterator：具体的迭代器实现对象。实现对聚合对象的遍历，并跟踪遍历时的当前位置。
- Aggregate：聚合对象。定义创建相应迭代器对象的接口。
- ConcreteAggregate：具体聚合对象。实现创建相应的迭代器对象。

14.2.3　迭代器模式示例代码

（1）先来看看迭代器接口的定义。示例代码如下：

```
/**
 * 迭代器接口，定义访问和遍历元素的操作
 */
```

```
public interface Iterator {
    /**
     * 移动到聚合对象的第一个位置
     */
    public void first();
    /**
     * 移动到聚合对象的下一个位置
     */
    public void next();
    /**
     * 判断是否已经移动到聚合对象的最后一个位置
     * @return true表示已经移动到聚合对象的最后一个位置
     *         false表示还没有移动到聚合对象的最后一个位置
     */
    public boolean isDone();
    /**
     * 获取迭代的当前元素
     * @return 迭代的当前元素
     */
    public Object currentItem();
}
```

（2）接下来看看具体的迭代器实现示意。示例代码如下：

```
/**
 * 具体的迭代器实现对象，示意的是聚合对象为数组的迭代器
 * 不同的聚合对象相应的迭代器实现是不一样的
 */
public class ConcreteIterator implements Iterator {
    /**
     * 持有被迭代的具体的聚合对象
     */
    private ConcreteAggregate aggregate;
    /**
     * 内部索引，记录当前迭代到的索引位置
     * -1表示刚开始的时候，迭代器指向聚合对象第一个对象之前
     */
    private int index = -1;
    /**
     * 构造方法，传入被迭代的具体的聚合对象
     * @param aggregate 被迭代的具体的聚合对象
     */
```

```
    public ConcreteIterator(ConcreteAggregate aggregate) {
        this.aggregate = aggregate;
    }

    public void first(){
        index = 0;
    }
    public void next(){
        if(index < this.aggregate.size()){
            index = index + 1;
        }
    }
    public boolean isDone(){
        if(index == this.aggregate.size()){
            return true;
        }
        return false;
    }
    public Object currentItem(){
        return this.aggregate.get(index);
    }
}
```

（3）再来看看聚合对象的定义。示例代码如下：

```
/**
 * 聚合对象的接口，定义创建相应迭代器对象的接口
 */
public abstract class Aggregate {
    /**
     * 工厂方法，创建相应迭代器对象的接口
     * @return 相应迭代器对象的接口
     */
    public abstract Iterator createIterator();
}
```

（4）下面来看看具体的聚合对象的实现，这里示意的是数组。示例代码如下：

```
/**
 * 具体的聚合对象，实现创建相应迭代器对象的功能
 */
public class ConcreteAggregate extends Aggregate {
    /**
```

```
     * 示意，表示聚合对象具体的内容
     */
    private String[] ss = null;

    /**
     * 构造方法，传入聚合对象具体的内容
     * @param ss 聚合对象具体的内容
     */
    public ConcreteAggregate(String[] ss){
        this.ss = ss;
    }

    public Iterator createIterator() {
        //实现创建Iterator的工厂方法
        return new ConcreteIterator(this);
    }
    /**
     *  获取索引所对应的元素
     * @param index 索引
     * @return 索引所对应的元素
     */
    public Object get(int index){
        Object retObj = null;
        if(index < ss.length){
            retObj = ss[index];
        }
        return retObj;
    }
    /**
     * 获取聚合对象的大小
     * @return 聚合对象的大小
     */
    public int size(){
        return this.ss.length;
    }
}
```

（5）最后来看看如何使用这个聚合对象和迭代器对象。示例代码如下：

```
public class Client {
    /**
     * 示意方法，使用迭代器的功能
```

```
     * 这里示意使用迭代器来迭代聚合对象
     */
    public void someOperation(){
        String[] names = {"张三","李四","王五"};
        //创建聚合对象
        Aggregate aggregate = new ConcreteAggregate(names);
        //循环输出聚合对象中的值
        Iterator it = aggregate.createIterator();
        //首先设置迭代器到第一个元素
        it.first();
        while(!it.isDone()){
            //取出当前的元素来
            Object obj = it.currentItem();
            System.out.println("the obj=="+obj);
            //如果还没有迭代到最后，那么就向下迭代一个
            it.next();
        }
    }
    public static void main(String[] args) {
        //可以简单地测试一下
        Client client = new Client();
        client.someOperation();
    }
}
```

14.2.4　使用迭代器模式来实现示例

要使用迭代器模式来实现示例，先来看看已有的两个工资系统现在的情况，然后再根据前面学习的迭代器模式来改造。

1. 已有的系统

（1）首先是有一个已经统一了的工资描述模型。为了演示简单，这里只留下最基本的字段，描述一下支付工资的人员、支付的工资数额，其他的包括时间等都不描述了；同时为了后面调试方便，实现了 toString 方法。示例代码如下：.

```
/**
 * 工资描述模型对象
 */
public class PayModel {
    /**
     * 支付工资的人员
```

```
     */
    private String userName;
    /**
     * 支付的工资数额
     */
    private double pay;
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public double getPay() {
        return pay;
    }
    public void setPay(double pay) {
        this.pay = pay;
    }
    public String toString(){
        return "userName="+userName+",pay="+pay;
    }
}
```

（2）客户方已有的工资管理系统中的工资管理类，内部是通过 List 来管理的。简单的示例代码如下：

```
/**
 * 客户方已有的工资管理对象
 */
public class PayManager{
    /**
     * 聚合对象，这里是Java的集合对象
     */
    private List list = new ArrayList();
    /**
     * 获取工资列表
     * @return 工资列表
     */
    public List getPayList(){
        return list;
    }
    /**
```

```
     * 计算工资，其实应该有很多参数，为了演示从简
     */
    public void calcPay(){
        //计算工资，并把工资信息填充到工资列表中
        //为了测试，输入些数据进去
        PayModel pm1 = new PayModel();
        pm1.setPay(3800);
        pm1.setUserName("张三");
                PayModel pm2 = new PayModel();
        pm2.setPay(5800);
        pm2.setUserName("李四");

        list.add(pm1);
        list.add(pm2);
    }
}
```

（3）客户方收购的那家公司的工资管理系统中的工资管理类，内部是通过数组来管理的。简单的示例代码如下：

```
/**
 * 被客户方收购的那个公司的工资管理类
 */
public class SalaryManager{
    /**
     * 用数组管理
     */
    private PayModel[] pms = null;
    /**
     * 获取工资列表
     * @return 工资列表
     */
    public PayModel[] getPays(){
        return pms;
    }
    /**
     * 计算工资，其实应该有很多参数，为了演示从简
     */
    public void calcSalary(){
        //计算工资，并把工资信息填充到工资列表中
        //为了测试，输入些数据进去
        PayModel pm1 = new PayModel();
```

```
        pm1.setPay(2200);
        pm1.setUserName("王五");

        PayModel pm2 = new PayModel();
        pm2.setPay(3600);
        pm2.setUserName("赵六");

        pms = new PayModel[2];
        pms[0] = pm1;
        pms[1] = pm2;
    }
}
```

（4）如果此时从外部来访问这两个工资列表，外部要采用不同的访问方式：一个是访问数组，另一个是访问集合对象。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //访问集团的工资列表
        PayManager payManager= new PayManager();
        //先计算再获取
        payManager.calcPay();
        Collection payList = payManager.getPayList();
        Iterator it = payList.iterator();
        System.out.println("集团工资列表：");
        while(it.hasNext()){
            PayModel pm = (PayModel)it.next();
            System.out.println(pm);
        }

        //访问新收购公司的工资列表
        SalaryManager salaryManager = new SalaryManager();
        //先计算再获取
        salaryManager.calcSalary();
        PayModel[] pms = salaryManager.getPays();
        System.out.println("新收购的公司工资列表：");
        for(int i=0;i<pms.length;i++){
            System.out.println(pms[i]);
        }
    }
}
```

仔细查看框住的代码，会发现它们的访问方式是完全不一样的。

运行结果如下：

```
集团工资列表：
userName=张三,pay=3800.0
userName=李四,pay=5800.0
新收购的公司工资列表：
userName=王五,pay=2200.0
userName=赵六,pay=3600.0
```

2. 统一访问聚合的接口

要使用迭代器模式来整合访问上面两个聚合对象，那就需要先定义出抽象的聚合对象和迭代器接口来，然后再提供相应的实现。

使用迭代器模式实现示例的结构如图 14.2 所示。

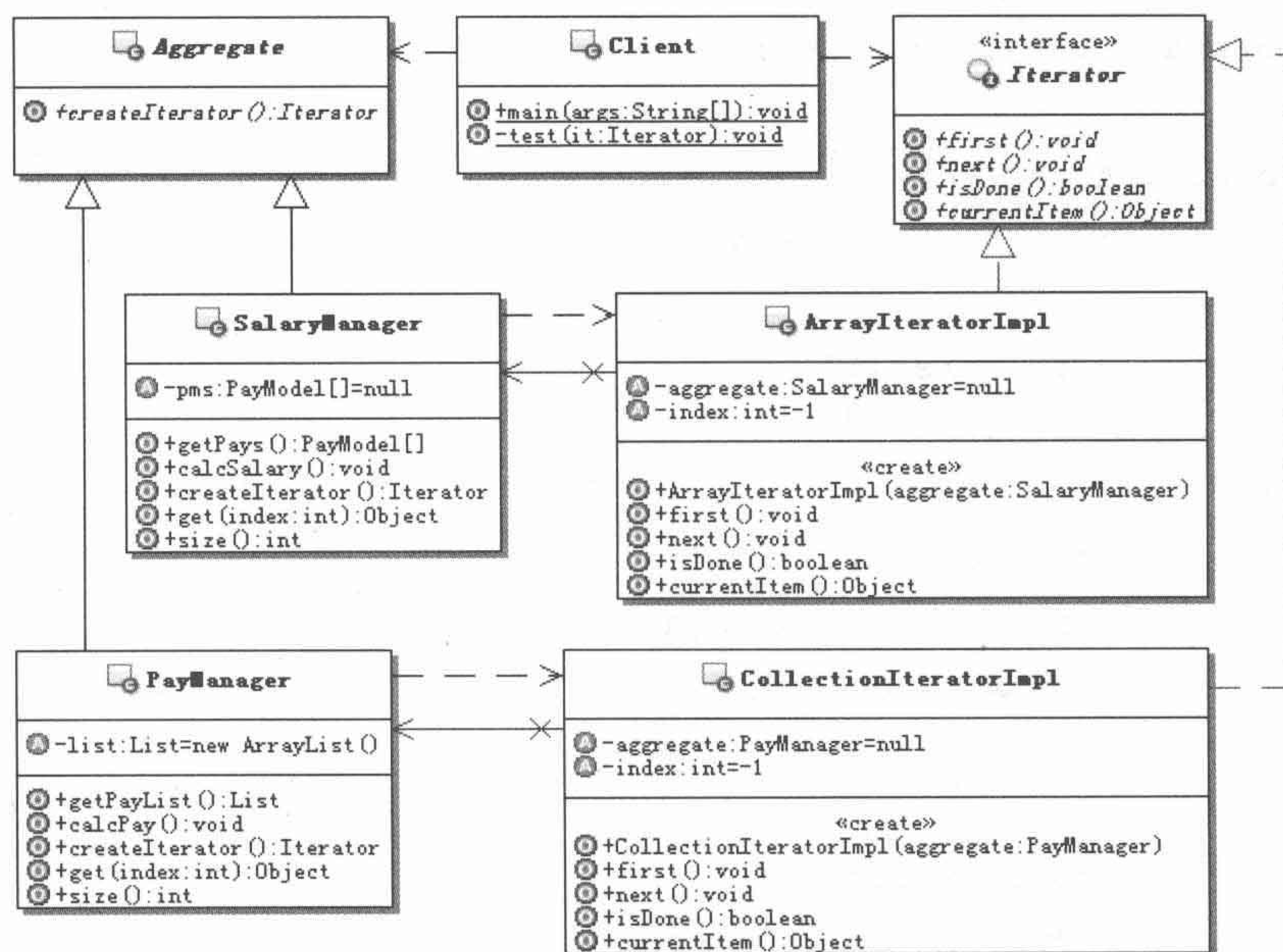


图 14.2　使用迭代器模式实现示例的结构示意图

（1）为了让客户端能够以一个统一的方式进行访问，最容易的方式就是为它们定义一个统一的接口，通过统一的接口来访问。这个示例用的 Iterator 和模式的示例代码是一样的，这里就不注释了。示例代码如下：

```
public interface Iterator {
    public void first();
    public void next();
    public boolean isDone();
    public Object currentItem();
}
```

（2）定义好了统一的接口，那就得分别实现这个接口。一个是 List 实现的，另一个

是数组实现的，先来看数组实现的访问。示例代码如下：

```
/**
 * 用来实现访问数组的迭代接口
 */
public class ArrayIteratorImpl implements Iterator{
    /**
     * 用来存放被迭代的聚合对象
     */
    private SalaryManager aggregate = null;
    /**
     * 用来记录当前迭代到的位置索引
     * -1表示刚开始的时候，迭代器指向聚合对象第一个对象之前
     */
    private int index = -1;

    public ArrayIteratorImpl(SalaryManager aggregate){
        this.aggregate = aggregate;
    }
    public void first(){
        index = 0;
    }
    public void next(){
        if(index < this.aggregate.size()){
            index = index + 1;
        }
    }
    public boolean isDone(){
        if(index == this.aggregate.size()){
            return true;
        }
        return false;
    }
    public Object currentItem(){
        return this.aggregate.get(index);
    }
}
```

为了让客户端能以统一的方式访问数据，所以对集合也提供一个对接口 Iterator 的实现。示例代码如下：

```
/**
```

```
 * 用来实现访问Collection集合的迭代接口，为了外部统一访问方式
 */
public class CollectionIteratorImpl implements Iterator{
    /**
     * 用来存放被迭代的聚合对象
     */
    private PayManager aggregate = null;
    /**
     * 用来记录当前迭代到的位置索引
     * -1表示刚开始的时候，迭代器指向聚合对象第一个对象之前
     */
    private int index = -1;

    public CollectionIteratorImpl(PayManager aggregate){
        this.aggregate = aggregate;
    }
    public void first(){
        index = 0;
    }
    public void next(){
        if(index < this.aggregate.size()){
            index = index + 1;
        }
    }
    public boolean isDone(){
        if(index == this.aggregate.size()){
            return true;
        }
        return false;
    }
    public Object currentItem(){
        return this.aggregate.get(index);
    }
}
```

（3）获取访问聚合的接口。

定义好了统一的访问聚合的接口，也分别实现了这个接口，新的问题是，在客户端如何才能获取这个访问聚合的接口呢？而且还要以统一的方式来获取。

一个简单的方案就是定义一个获取访问聚合的接口的接口，客户端先通过这个接口来获取访问聚合的接口，然后再访问聚合对象。示例代码如下：

```
public abstract class Aggregate {
    /**
     * 工厂方法，创建相应迭代器对象的接口
     * @return 相应迭代器对象的接口
     */
    public abstract Iterator createIterator();
}
```

然后让具体的聚合对象 PayManger 和 SalaryManager 来继承这个抽象类，提供分别访问它们的访问聚合的接口。

修改 PayManager 对象，添加 createIterator 方法的实现，另外再添加迭代器回调聚合对象的方法，一个方法是获取聚合对象的大小，另一个方法是根据索引获取聚合对象中的元素。示例代码如下：

```
public class PayManager extends Aggregate{
    public Iterator createIterator(){
        return new CollectionIteratorImpl(this);
    }
    public Object get(int index){
        Object retObj = null;
        if(index < this.list.size()){
            retObj = this.list.get(index);
        }
        return retObj;
    }
    public int size(){
        return this.list.size();
    }
}
```

其他的代码没有变化，为了节省篇幅，就省略了，上面的方法是新加的

同理修改 SalaryManager 对象。示例代码如下：

```
public class SalaryManager extends Aggregate{
    public Iterator createIterator(){
        return new ArrayIteratorImpl(this);
    }
    public Object get(int index){
        Object retObj = null;
        if(index < pms.length){
            retObj = pms[index];
        }
```

```
        return retObj;
    }
    public int size(){
        return this.pms.length;
    }
}
```

其他的代码没有变化，为了节省篇幅，就省略了，上面的方法是新加的

（4）统一访问的客户端。

下面就来看看客户端是如何通过迭代器接口来访问聚合对象的。为了显示是统一的访问，干脆把通过访问聚合的接口来访问聚合对象的功能独立成一个方法。虽然是访问不同的聚合对象，但是都调用这个方法去访问。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //访问集团的工资列表
        PayManager payManager= new PayManager();
        //先计算再获取
        payManager.calcPay();
        System.out.println("集团工资列表：");
        test(payManager.createIterator());

        //访问新收购公司的工资列表
        SalaryManager salaryManager = new SalaryManager();
        //先计算再获取
        salaryManager.calcSalary();
        System.out.println("新收购的公司工资列表：");
        test(salaryManager.createIterator());
    }

    /**
     * 测试通过访问聚合对象的迭代器，是否能正常访问聚合对象
     * @param it 聚合对象的迭代器
     */
    private static void test(Iterator it){
        //循环输出聚合对象中的值
        //首先设置迭代器到第一个元素
        it.first();
        while(!it.isDone()){
            //取出当前的元素来
```

调用统一的方法来访问聚合对象PayManager

调用统一的方法来访问聚合对象 SalaryManager

统一的通过迭代器接口来访问聚合对象的方法

```
                Object obj = it.currentItem();
                System.out.println("the obj=="+obj);
                //如果还没有迭代到最后，那么就向下迭代一个
                it.next();
            }
        }
}
```

运行一下客户端，测试看看效果。

> 提示
> **估计有些朋友看到这里，会觉得上面的实现特麻烦，会认为“Java 里面就有 Iterator 接口，而且 Java 集合框架中的聚合对象也大都实现了 Iterator 接口的功能，还有必要像上面这么做吗？”**
>
> **其实这么做，是为了让大家看到迭代器模式的全貌，后面会讲到用 Java 中的迭代器来实现。另外，有些时候还是需要自己来扩展和实现迭代器模式的，所以还是应该先独立学习迭代器模式。**

（5）迭代器示例小结。

如同前面的示例，提供了一个统一访问聚合对象的接口，通过这个接口就可以顺序地访问聚合对象的元素。对于客户端而言，只是面向这个接口在访问，根本不知道聚合对象内部的表示方法。

事实上，前面的例子故意做了一个集合类型的聚合对象和一个数组类型的聚合对象，但是从客户端来看，访问聚合的代码是完全一样的，根本看不出任何的差别，也看不出到底聚合对象内部是什么类型。

14.3 模 式 讲 解

14.3.1 认识迭代器模式

1. 迭代器模式的功能

迭代器模式的功能主要在于提供对聚合对象的迭代访问。迭代器就围绕着这个“访问”做文章，延伸出很多的功能来。比如：

- 以不同的方式遍历聚合对象，比如向前、向后等。
- 对同一个聚合同时进行多个遍历。
- 以不同的遍历策略来遍历聚合，比如是否需要过滤等。
- 多态迭代，含义是：为不同的聚合结构提供统一的迭代接口，也就是说通过一个迭代接口可以访问不同的聚合结构，这就叫做多态迭代。上面的示例就已经实现了多态迭代。事实上，标准的迭代模式实现基本上都是支持多态迭代的。

> 注意 **但是请注意：多态迭代可能会带来类型安全的问题，可以考虑使用泛型。**

2. 迭代器模式的关键思想

聚合对象的类型很多，如果对聚合对象的迭代访问跟聚合对象本身融合在一起的话，会严重影响到聚合对象的可扩展性和可维护性。

因此**迭代器模式的关键思想就是把对聚合对象的遍历和访问从聚合对象中分离出来，放入单独的迭代器中**，这样聚合对象会变得简单一些；而且迭代器和聚合对象可以独立地变化和发展，会大大加强系统的灵活性。

3. 内部迭代器和外部迭代器

所谓内部迭代器，指的是由迭代器自己来控制迭代下一个元素的步骤，客户端无法干预。因此，如果想要在迭代的过程中完成工作的话，客户端就需要把操作传递给迭代器。迭代器在迭代的时候会在每个元素上执行这个操作，类似于 Java 的回调机制。

所谓外部迭代器，指的是由客户端来控制迭代下一个元素的步骤，像前面的示例一样，客户端必须显式地调用 next 来迭代下一个元素。

总体来说外部迭代器比内部迭代器要灵活一些，因此我们常见的实现多属于外部迭代器。前面的例子也是实现的外部迭代器。

4. Java 中最简单的统一访问聚合的方式

如果只是想要使用一种统一的访问方式来访问聚合对象，在 Java 中有更简单的方式，简单到几乎什么都不用做，利用 Java 5 以上版本本身的特性即可。

> 注意 **请注意，这只是从访问形式上一致了，却也暴露了聚合的内部实现，因此并不能算是标准迭代器模式的实现，但是从某种意义上说，可以算是隐含地实现了部分迭代器模式的功能。**

那么怎么做呢？

为了简单，让我们回到没有添加任何迭代器模式的情况下。很简单，只要让聚合对象中的结合实现泛型即可。示例如下：

```
public class PayManager{
    private List<PayModel> list = new ArrayList<PayModel>();
    /**
     * 获取工资列表
     * @return 工资列表
     */
    public List<PayModel> getPayList(){
        return list;
    }
}
```

（这里改成用泛型，当然返回给客户端的也需要泛型）

（别的实现代码没有变化，为节省篇幅就省略了）

这样一来，客户端的代码就可以改成使用增强的 for 循环来实现了，对于数组、泛型的集合都可以采用一样的方法来实现了，从代码层面上看，就算是统一了访问聚合的方式了。修改后的客户端代码如下：

```
public class Client {
    public static void main(String[] args) {
        //访问集团的工资列表
        PayManager payManager= new PayManager();
        //先计算再获取
        payManager.calcPay();

        Collection<PayModel> payList = payManager.getPayList();
        System.out.println("集团工资列表：");

        //      Iterator it = payList.iterator();
        //      while(it.hasNext()){
        //          PayModel pm = (PayModel)it.next();
        //          System.out.println(pm);
        //      }

                for(PayModel pm : payList){
                    System.out.println(pm);
                }
        //访问新收购公司的工资列表
        SalaryManager salaryManager = new SalaryManager();
        //先计算再获取
        salaryManager.calcSalary();
        PayModel[] pms = salaryManager.getPays();
        System.out.println("新收购的公司工资列表：");
        //      for(int i=0;i<pms.length;i++){
        //          System.out.println(pms[i]);
        //      }
                for(PayModel pm : pms){
                System.out.println(pm);
                }
    }
}
```

这些是旧的访问实现

这两段新的访问方式是否一样呢？

这些是旧的访问实现

14.3.2 使用 Java 的迭代器

大家都知道，在 java.util 包里面有一个 Iterator 的接口，在 Java 中实现迭代器模式是

非常简单的，而且 Java 的集合框架中的聚合对象基本上都是提供了迭代器的。

下面就来把前面的例子改成用 Java 中的迭代器实现，一起来看看有些什么改变。

- 不再需要自己实现的 Iterator 接口，直接实现 java.util.Iterator 接口就可以了。所有使用自己实现的 Iterator 接口的地方都需要修改过来。
- Java 中 Iterator 接口跟前面自己定义的接口相比，需要实现的方法是不一样的。
- 集合已经提供了 Iterator，那么 CollectionIteratorImpl 类就不需要了，直接删除。

好了，还是一起来看看代码吧。

（1）PayModel 类没有任何变化，就不再示例了。

（2）抽象的 Aggregate 类就是把创建迭代器方法返回的类型转换成 Java 中的 Iterator 了。示例代码如下：

```
import java.util.Iterator;
public abstract class Aggregate {
    public abstract Iterator createIterator();
}
```

（3）原来的 ArrayIteratorImpl 类，实现的接口改变了，实现的代码也需要随着改变。示例代码如下：

```
/**
 * 用来实现访问数组的迭代接口
 */
public class ArrayIteratorImpl implements Iterator{
    /**
     * 用来存放被迭代的聚合对象
     */
    private SalaryManager aggregate = null;
    /**
     * 用来记录当前迭代到的位置索引
     */
    private int index = 0;
    public ArrayIteratorImpl(SalaryManager aggregate){
        this.aggregate = aggregate;
    }

    public boolean hasNext() {
        //判断是否还有下一个元素
        if(aggregate!=null && index<aggregate.size()){
            return true;
        }
        return false;
```

```
    }
    public Object next() {
        Object retObj = null;
        if(hasNext()){
            retObj = aggregate.get(index);
            //每取走一个值，就把已访问索引加1
            index++;
        }
        return retObj;
    }
    public void remove() {
        //暂时可以不实现
    }
}
```

（4）对于 PayManager 类，在实现创建迭代器的方法上发生了改变，不再使用自己实现的迭代器，改成 Java 的集合框架实现的迭代器了。示例代码如下：

```
public class PayManager extends Aggregate{
    private List<PayModel> list = new ArrayList<PayModel>();
    public List<PayModel> getPayList(){
        return list;
    }
    public void calcPay(){
        //计算工资，并把工资信息填充到工资列表中
        //为了测试，输入些点数据进去
        PayModel pm1 = new PayModel();
        pm1.setPay(3800);
        pm1.setUserName("张三");

        PayModel pm2 = new PayModel();
        pm2.setPay(5800);
        pm2.setUserName("李四");

        list.add(pm1);
        list.add(pm2);
    }
    public Iterator createIterator() {
        return list.iterator();
    }
}
```

直接使用集合框架提供的 Iterator 了

（5）对于 SalaryManager 类，除了创建迭代器方法返回的类型改变外，其他的都没有改变，还是用 ArrayIteratorImpl 来实现迭代器。

（6）接下来写个客户端来测试看看。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //访问集团的工资列表
        PayManager payManager= new PayManager();
        //先计算再获取
        payManager.calcPay();
        System.out.println("集团工资列表：");
        test(payManager.createIterator());

        //访问新收购公司的工资列表
        SalaryManager salaryManager = new SalaryManager();
        //先计算再获取
        salaryManager.calcSalary();
        System.out.println("新收购的公司工资列表：");
        test(salaryManager.createIterator());
    }
    /**
     * 测试通过访问聚合对象的迭代器，是否能正常访问聚合对象
     * @param it 聚合对象的迭代器
     */
    private static void test(Iterator it){
        while(it.hasNext()){
            PayModel pm = (PayModel)it.next();
            System.out.println(pm);
        }
    }
}
```

很明显，改用 Java 的 Iterator 来实现，比自己全部重新去做，还是要简单一些的。

14.3.3　带迭代策略的迭代器

由于迭代器模式把聚合对象和访问聚合的机制实现了分离，所以可以在迭代器上实现不同的迭代策略，最为典型的就是实现过滤功能的迭代器。

延伸 **在实际开发中，对于经常被访问的一些数据可以使用缓存，把这些数据存放在内存中。但是不同的业务功能需要访问的数据是不同的，还有不同的业务访问权限能访问的数据也是不同的。对于这种情况，就可以使用实现过滤功能的迭代器，让不同功能使用不同的迭代器来访问。当然，这种情况也可以结合策略模式来实现。**

在实现过滤功能的迭代器中，又有两种常见的需要过滤的情况，一种是对数据进行整条过滤，比如只能查看自己部门的数据；另外一种情况是对数据进行部分过滤，比如某些人不能查看工资数据。

带迭代策略的迭代器实现的一个基本思路，就是先把聚合对象的聚合数据获取到，并存储到迭代器中，这样迭代器就可以按照不同的策略来迭代数据了。

1. 带迭代策略的迭代器示例

沿用上一个例子，来修改 ArrayIteratorImpl 简单地示意一下，不考虑复杂的算法。大致的修改如下。

- 原来是持有聚合对象的，现在直接把这个聚合对象的内容取出来存放到迭代器中。也就是迭代的时候，直接在迭代器中获取具体的聚合对象的元素，这样才好控制迭代的数据。
- 在迭代器的具体实现中加入过滤的功能。

示例代码如下：

```
/**
 * 用来实现访问数组的迭代接口，加入了迭代策略
 */
public class ArrayIteratorImpl implements Iterator{
    /**
     * 用来存放被迭代的数组
     */
    private PayModel[] pms = null;
    /**
     * 用来记录当前迭代到的位置索引
     */
    private int index = 0;

    public ArrayIteratorImpl(SalaryManager aggregate){
        //在这里先对聚合对象的数据进行过滤，比如工资必须在3000以下
        Collection<PayModel> tempCol = new ArrayList<PayModel>();
        for(PayModel pm : aggregate.getPays()){
            if(pm.getPay() < 3000){
                tempCol.add(pm);
```

对数据进行整条过滤

```
            }
        }
        //然后把符合要求的数据存放到用来迭代的数组
        this.pms = new PayModel[tempCol.size()];
        int i=0;
        for(PayModel pm : tempCol){
            this.pms[i] = pm;
            i++;
        }
    }
    public boolean hasNext() {
        //判断是否还有下一个元素
        if(pms!=null && index<=(pms.length-1)){
            return true;
        }
        return false;
    }
    public Object next() {
        Object retObj = null;
        if(hasNext()){
            retObj = pms[index];
            //每取走一个值，就把已访问索引加1
            index++;
        }
        //在这里对要返回的数据进行过滤，比如不让查看工资数据
        ((PayModel)retObj).setPay(0.0);

        return retObj;
    }
    public void remove() {
        //暂时可以不实现
    }
}
```

对数据进行部分过滤

2. 谁定义遍历算法的问题

在实现迭代器模式的时候，一个常见的问题就是：谁来定义遍历算法？其实带策略的迭代器讲述的也是这个问题。

在迭代器模式的实现中，常见的有两个地方可以来定义遍历算法，一个是聚合对象本身，另外一个就是迭代器负责遍历算法。

在聚合对象本身定义遍历算法的这种情况下，通常会在遍历过程中，用迭代器来存

储当前迭代的状态，这种迭代器被称为**游标**，因为它仅用来指示当前的位置。比如在14.2.4节中示例的迭代器就属于这种情况。

在迭代器中定义遍历算法，会比在相同的聚合上使用不同的迭代算法容易，同时也易于在不同的聚合上重用相同的算法。比如上面带策略的迭代器的示例，迭代器把需要迭代的数据从聚合对象中取出并存放到自己的对象中，然后再迭代自己的数据，这样一来，除了刚开始创建迭代器的时候需要访问聚合对象外，真正迭代过程已经跟聚合对象无关了。

但是，在迭代器中定义遍历算法，如果实现遍历算法需要访问聚合对象的私有变量，那么将遍历算法放入迭代器中会破坏聚合对象的封装性。

至于究竟使用哪一种方式，要具体问题具体分析。

14.3.4 双向迭代器

所谓双向迭代器的意思就是：**可以同时向前和向后遍历数据的迭代器。**

在Java util包中的ListIterator接口就是一个双向迭代器的示例。当然自己实现双向迭代器也非常容易，只要在自己的Iterator接口中添加向前的判断和向前获取值的方法，然后在实现中实现即可。

延续14.2.4节的示例，来自己实现双向迭代器，相同的部分就不再示范了，只演示不同的地方。

先看看新的迭代器接口。示例代码如下：

```
/**
 * 迭代器接口，定义访问和遍历元素的操作，实现双向迭代
 */
public interface Iterator {
    public void first();
    public void next();                    已有的定义
    public boolean isDone();
    public Object currentItem();
    /**
     * 判断是否为第一个元素
     * @return 如果为第一个元素，返回true，否则返回false
     */
    public boolean isFirst();
    /**
     * 移动到聚合对象的上一个位置
     */
    public void previous();
}
```

有了新的迭代器接口，也应该有新的实现。示例代码如下：

```
/**
 * 用来实现访问数组的双向迭代接口
 */
public class ArrayIteratorImpl implements Iterator{
    private SalaryManager aggregate = null;
    private int index = -1;
    public ArrayIteratorImpl(SalaryManager aggregate){
        this.aggregate = aggregate;
    }
    public void first(){
        index = 0;
    }
    public void next(){
        if(index < this.aggregate.size()){
            index = index + 1;
        }
    }
    public boolean isDone(){
        if(index == this.aggregate.size()){
            return true;
        }
        return false;
    }
    public Object currentItem(){
        return this.aggregate.get(index);
    }

    public boolean isFirst(){
        if(index==0){
            return true;
        }
        return false;
    }
    public void previous(){
        if(index > 0 ){
            index = index - 1;
        }
    }
}
```

原有的实现，没有改变

基本实现完了，写个客户端来享受一下双向迭代的乐趣。由于这个实现要考虑同时控制向前和向后迭代取值，而控制当前索引的是同一个值，因此在获取向前取值的时候，要先把已访问索引减去1，然后再取值，这个跟向后取值是反过来的，注意一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //访问新收购公司的工资列表
        SalaryManager salaryManager = new SalaryManager();
        //先计算再获取
        salaryManager.calcSalary();

        //得到双向迭代器
        Iterator it = salaryManager.createIterator();
        //首先设置迭代器到第一个元素
        it.first();

        //先next一个
        if(!it.isDone()){
            PayModel pm = (PayModel)it.currentItem();
            System.out.println("next1 == "+pm);
            //向下迭代一个
            it.next();
        }
        //然后previous一个
        if(!it.isFirst()){
            //向前迭代一个
            it.previous();
            PayModel pm = (PayModel)it.currentItem();
            System.out.println("previous1 == "+pm);
        }
        //再next一个
        if(!it.isDone()){
            PayModel pm = (PayModel)it.currentItem();
            System.out.println("next2 == "+pm);
            //向下迭代一个
            it.next();
        }
        //继续next一个
        if(!it.isDone()){
            PayModel pm = (PayModel)it.currentItem();
```

注意这里是先移动位置，然后再取值

```
            System.out.println("next3 == "+pm);
            //向下迭代一个
            it.next();
        }
        //然后previous一个
        if(!it.isFirst()){
            //向前迭代一个
            it.previous();
            PayModel pm = (PayModel)it.currentItem();
            System.out.println("previous2 == "+pm);
        }

    }
}
```

上面的示例故意先向后取值，然后再向前取值，这样反复才能看出双向迭代器的效果。运行结果如下：

```
next1 == userName=王五,pay=2200.0
previous1 == userName=王五,pay=2200.0
next2 == userName=王五,pay=2200.0
next3 == userName=赵六,pay=3600.0
previous2 == userName=赵六,pay=3600.0
```

可能有些人会疑惑：为什么 next1 和 previous1 取出来的值是一样的呢？

这是因为现在是顺序迭代，当 next 显示第一条的时候，内部索引已经指向第二条了，所以这个时候再 previous 向前一条的时候，数据就是第一条数据了。

再仔细查看上面的结果，发现这个时候继续 next 数据时，数据还是第一条数据，同理，刚才 previous 向前一条的时候，内部索引已经指向第一条之前了。

14.3.5　迭代器模式的优点

- 更好的封装性
- 迭代器模式可以让你访问一个聚合对象的内容，而无须暴露该聚合对象的内部表示，从而提高聚合对象的封装性。
- 可以以不同的遍历方式来遍历一个聚合
- 使用迭代器模式，使得聚合对象的内容和具体的迭代算法分离开。这样就可以通过使用不同的迭代器的实例、不同的遍历方式来遍历一个聚合对象了，比如上面示例的带迭代策略的迭代器。
- 迭代器简化了聚合的接口
- 有了迭代器的接口，则聚合本身就不需要再定义这些接口了，从而简化了聚合的接口定义。

- 简化客户端调用
- 迭代器为遍历不同的聚合对象提供了一个统一的接口，使得客户端遍历聚合对象的内容变得更简单。
- 同一个聚合上可以有多个遍历
- 每个迭代器保持它自己的遍历状态，比如前面实现中的迭代索引位置，因此可以对同一个聚合对象同时进行多个遍历。

14.3.6 思考迭代器模式

1. 迭代器模式的本质

> 迭代器模式的本质：**控制访问聚合对象中的元素。**

迭代器能实现“无须暴露聚合对象的内部实现，就能够访问到聚合对象中的各个元素”的功能，看起来其本质应该是“**透明**访问聚合对象中的元素”。

> 提示 **但仔细思考一下，除了透明外，迭代器就没有别的功能了吗？很明显还有其他的功能，前面也讲到了一些，比如“带迭代策略的迭代器”。那么综合来看，迭代器模式的本质应该是“控制访问聚合对象中的元素”，而非单纯的“透明”。事实上，“透明”访问也是“控制访问”的一种情况。**

认识这个本质，对于识别和变形使用迭代器模式很有帮助。大家想想，现在的迭代模式默认的都是向前或者向后获取一个值，也就是说都是单步迭代，那么，如果想要控制一次迭代多条怎么办呢？

这个在实际开发中是很有用的，比如在实际开发中常用的翻页功能的实现。翻页功能有如下几种实现方式。

（1）纯数据库实现。

依靠 SQL 提供的功能实现翻页，用户每次请求翻页的数据，就会到数据库中获取相应的数据。

（2）纯内存实现。

就是一次性从数据库中把需要的所有数据都取出来放到内存中，然后用户请求翻页时，从内存中获取相应的数据。

上面两种方案各有优缺点：

第一种方案明显是时间换空间的策略，每次获取翻页的数据都要访问数据库，运行速度相对比较慢，而且很耗数据库资源，但是节省了内存空间；

第二种方案是典型的空间换时间，每次是直接从内存中获取翻页的数据，运行速度快，但是很耗内存。

在实际开发中，小型系统一般采用第一种方案，基本没有单独采用第二种方案的，因为内存实在是太宝贵了；中大型的系统一般是把两个方案结合起来，综合利用它们的优点，而又规避它们的缺点，从而更好地实现翻页的功能。

（3）纯数据库实现 + 纯内存实现。

思路是这样的：如果每页显示 10 条记录，根据判断，用户很少翻到 10 页以后，那好，第一次访问的时候，就一次性从数据库中获取前 10 页的数据，也就是 100 条记录，把这 100 条记录放在内存里面。

这样一来，当用户在前 10 页内进行翻页操作的时候，就不用再访问数据库了，而是直接从内存中获取数据，速度就快了。

当用户想要获取第 11 页的数据，这个时候才会再次访问数据库。对于这个时候到底获取多少页的数据，简单的处理就是继续获取 10 页的数据。比较好的方式就是根据访问统计进行衰减访问，比如折半获取，也就是第一次访问数据库获取 10 页的数据，那么第二次就只获取 5 页，如此操作直到一次从数据库中获取一页的数据。这也符合正常规律，因为越到后面，被用户翻页到的机会也就越小了。

对于翻页的迭代，后面将给大家一个简单的示例。

2. 何时选用迭代器模式

建议在以下情况中选用迭代器模式。

- 如果你希望提供访问一个聚合对象的内容，但是又不想暴露它的内部表示的时候，可以使用迭代器模式来提供迭代器接口，从而让客户端只是通过迭代器的接口来访问聚合对象，而无须关心聚合对象的内部实现。
- 如果你希望有多种遍历方式可以访问聚合对象，可以使用迭代器模式。
- 如果你希望为遍历不同的聚合对象提供一个统一的接口，可以使用迭代器模式。

14.3.7　翻页迭代

在上面讲到的翻页实现机制中，只要使用到内存来缓存数据，就涉及到翻页迭代的实现。简单点说，就是一次迭代，会要求迭代取出一页的数据，而不是一条数据。

其实实现翻页迭代也很简单，主要是把原来一次迭代一条数据的接口，都修改成一次迭代一页的数据就可以了。**在具体的实现上，又分成顺序翻页迭代器和随机翻页迭代器。**

1. 顺序翻页迭代器示例

（1）先看看迭代器接口的定义。示例代码如下：

```
/**
 * 定义翻页访问聚合元素的迭代接口
 */
public interface AggregationIterator {
```

```
    /**
     * 判断是否还有下一个元素，无所谓是否够一页的数据
     * 因为最后哪怕只有一条数据，也是要算一页的
     * @return 如果有下一个元素，返回true，没有下一个元素就返回false
     */
    public boolean hasNext();
    /**
     * 取出下面几个元素
     * @param num 需要获取的记录条数
     * @return 下面几个元素
     */
    public Collection next(int num);
    /**
     * 判断是否还有上一个元素，无所谓是否够一页的数据
     * 因为最后哪怕只有一条数据，也是要算一页的
     * @return 如果有上一个元素，返回true，没有上一个元素就返回false
     */
    public boolean hasPrevious();
    /**
     * 取出上面几个元素
     * @param num 需要获取的记录条数
     * @return 上面几个元素
     */
    public Collection previous(int num);
}
```

（2）PayModel 和前面的示例是一样的，这里就不再赘述。

（3）接下来看看 SalaryManager 的实现，有如下改变。

- 不用再实现获取聚合对象大小和根据索引获取聚合对象中的元素的功能了。
- 在准备测试数据的时候，多准备几条，方便看出翻页的效果。

示例代码如下：

```
/**
 * 被客户方收购的那个公司的工资管理类
 */
public class SalaryManager{
    private PayModel[] pms = null;
    public PayModel[] getPays(){
        return pms;
    }
    public void calcSalary(){
```

```
        //计算工资，并把工资信息填充到工资列表中
        //为了测试，输入些数据进去
        PayModel pm1 = new PayModel();
        pm1.setPay(2200);
        pm1.setUserName("王五");

        PayModel pm2 = new PayModel();
        pm2.setPay(3600);
        pm2.setUserName("赵六");

        PayModel pm3 = new PayModel();
        pm3.setPay(2200);
        pm3.setUserName("王五二号");

        PayModel pm4 = new PayModel();
        pm4.setPay(3600);
        pm4.setUserName("赵六二号");

        PayModel pm5 = new PayModel();
        pm5.setPay(2200);
        pm5.setUserName("王五三号");

        pms = new PayModel[5];
        pms[0] = pm1;
        pms[1] = pm2;
        pms[2] = pm3;
        pms[3] = pm4;
        pms[4] = pm5;
    }
    public AggregationIterator createIterator() {
        return new ArrayIteratorImpl(this);
    }
}
```

（4）来看看如何实现迭代器接口。示例代码如下：

```
/**
 * 用来实现翻页访问聚合元素的迭代接口
 */
public class ArrayIteratorImpl implements AggregationIterator{
    /**
     * 用来存放被迭代的数组
```

```
     */
    private PayModel[] pms = null;
     /**
      * 用来记录当前迭代到的位置索引
      */
    private int index = 0;
    public ArrayIteratorImpl(SalaryManager aggregate){
        this.pms = aggregate.getPays();
    }
    public boolean hasNext() {
        //判断是否还有下一个元素
        if(pms!=null && index<=(pms.length-1)){
            return true;
        }
        return false;
    }
public boolean hasPrevious() {
        if(pms!=null && index > 0){
            return true;
        }
        return false;
    }
    public Collection next(int num) {
        Collection col = new ArrayList();
        int count=0;
        while(hasNext() && count<num){
            col.add(pms[index]);
            //每取走一个值，就把已访问索引加1
            index++;
            count++;
        }
        return col;
    }
    public Collection previous(int num){
        Collection col = new ArrayList();
        int count=0;
        //简单的实现就是把索引退回去num个，然后再取值
        //但事实上这种实现是有可能多退回去数据的，比如，已经到了最后一页
        //而且最后一页的数据不够一页的数据，那么退回去num个索引就退多了
        //为了示例的简洁性，这里就不去处理了
```

现在需要返回多条记录的集合了，也不再是 if，而是 while

```
        index = index - num;
        while(hasPrevious() && count<num){
            col.add(pms[index]);
            index ++;
            count++;
        }
        return col;
    }
}
```

（5）写个客户端测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //访问新收购公司的工资列表，先计算再获取
        SalaryManager salaryManager = new SalaryManager();
        salaryManager.calcSalary();
        //得到翻页迭代器
        AggregationIterator it = salaryManager.createIterator();

        //获取第一页，每页显示2条
        Collection col = it.next(2);
        System.out.println("第一页数据：");
        print(col);
        //获取第二页，每页显示2条
        Collection col2 = it.next(2);
        System.out.println("第二页数据：");
        print(col2);

        //向前一页，也就是再次获取第二页
        Collection col3 = it.previous(2);
        System.out.println("再次获取第二页数据：");
        print(col3);
    }
    private static void print(Collection col){
        Iterator it = col.iterator();
        while(it.hasNext()){
            Object obj = it.next();
            System.out.println(obj);
        }
    }
}
```

输出集合中的值，测试用

运行结果如下：

```
第一页数据：
userName=王五,pay=2200.0
userName=赵六,pay=3600.0
第二页数据：
userName=王五二号,pay=2200.0
userName=赵六二号,pay=3600.0
再次获取第二页数据：
userName=王五二号,pay=2200.0
userName=赵六二号,pay=3600.0
```

仍然是顺序迭代的，也就是获取完第二页数据，内部索引就指向后面了，这个时候再运行向前一页，取的就还是第二页的数据了。

2. 随机翻页迭代器示例

估计看到这里，有些朋友会想，实际应用中，用户怎么会这么老实，按照顺序访问？通常情况都是随意的访问页数，比如看了第一页可能就直接看第三页了，看完第三页他又想看第一页。

这就需要随机翻页迭代器了，也就是可以指定页面号和每页显示的数据来访问数据的迭代器。下面来看看示例。

（1）修改迭代接口的方法。不需要再有向前和向后的方法，取而代之的是指定页面号和每页显示的数据来访问的方法。示例代码如下：

```
/**
 * 定义随机翻页访问聚合元素的迭代接口
 */
public interface AggregationIterator {
    /**
     * 判断是否还有下一个元素，无所谓是否够一页的数据
     * 因为最后哪怕只有一条数据，也是要算一页的
     * @return 如果有下一个元素，返回true，没有下一个元素就返回false
     */
    public boolean hasNext();
    /**
     * 判断是否还有上一个元素，无所谓是否够一页的数据
     * 因为最后哪怕只有一条数据，也是要算一页的
     * @return 如果有上一个元素，返回true，没有上一个元素就返回false
     */
    public boolean hasPrevious();
    /**
     * 取出指定页数的数据
     * @param pageNum 要获取的页数
```

```
     * @param pageShow 每页显示的数据条数
     * @return 指定页数的数据
     */
    public Collection getPage(int pageNum,int pageShow);
}
```

（2）定义了接口，看看具体的实现。示例代码如下：

```
/**
 * 用来实现随机翻页访问聚合元素的迭代接口
 */
public class ArrayIteratorImpl implements AggregationIterator{
    /**
     * 用来存放被迭代的数组
     */
    private PayModel[] pms = null;
    /**
     * 用来记录当前迭代到的位置索引
     */
    private int index = 0;
    public ArrayIteratorImpl(SalaryManager aggregate){
        this.pms = aggregate.getPays();
    }
    public boolean hasNext() {
        //判断是否还有下一个元素
        if(pms!=null && index<=(pms.length-1)){
            return true;
        }
        return false;
    }
    public boolean hasPrevious() {
        if(pms!=null && index > 0){
            return true;
        }
        return false;
    }
    public Collection getPage(int pageNum,int pageShow){
        Collection col = new ArrayList();
        //需要在这里先计算需要获取的数据的开始条数和结束条数
        int start = (pageNum-1)*pageShow;
        int end = start + pageShow-1;
        //控制start的边界，最小是0
```

```
        if(start < 0){
            start = 0;
        }
        //控制end的边界，最大是数组的最大索引
        if(end > this.pms.length-1){
            end = this.pms.length - 1;
        }
        //每次取值都是从头开始循环，所以设置index为0
        index = 0;
        while(hasNext()  && index<=end){
            if(index >= start){
                col.add(pms[index]);
            }
            //把已访问索引加1
            index++;
        }
        return col;
    }
}
```

（3）写个客户端，测试看看，是否能实现随机的翻页。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //访问新收购公司的工资列表
        SalaryManager salaryManager = new SalaryManager();
        //先计算再获取
        salaryManager.calcSalary();
        //得到翻页迭代器
        AggregationIterator it = salaryManager.createIterator();

        //获取第一页，每页显示2条
        Collection col = it.getPage(1,2);
        System.out.println("第一页数据：");
        print(col);
        //获取第二页，每页显示2条
        Collection col2 = it.getPage(2,2);
        System.out.println("第二页数据：");
        print(col2);
        //再次获取第一页
        Collection col3 = it.getPage(1,2);
        System.out.println("再次获取第一页数据：");
```

```
        print(col3);
        //获取第三页
        Collection col4 = it.getPage(3,2);
        System.out.println("获取第三页数据：");
        print(col4);
    }
    private static void print(Collection col){
        Iterator it = col.iterator();
        while(it.hasNext()){
            Object obj = it.next();
            System.out.println(obj);
        }
    }
}
```

测试结果如下：

```
第一页数据：
userName=王五,pay=2200.0
userName=赵六,pay=3600.0
第二页数据：
userName=王五二号,pay=2200.0
userName=赵六二号,pay=3600.0
再次获取第一页数据：
userName=王五,pay=2200.0
userName=赵六,pay=3600.0
获取第三页数据：
userName=王五三号,pay=2200.0
```

14.3.8 相关模式

- 迭代器模式和组合模式

  这两个模式可以组合使用。

  组合模式是一种递归的对象结构，在枚举某个组合对象的子对象的时候，通常会使用迭代器模式。

- 迭代器模式和工厂方法模式

  这两个模式可以组合使用。

  在聚合对象创建迭代器的时候，通常会采用工厂方法模式来实例化相应的迭代器对象。

读书笔记

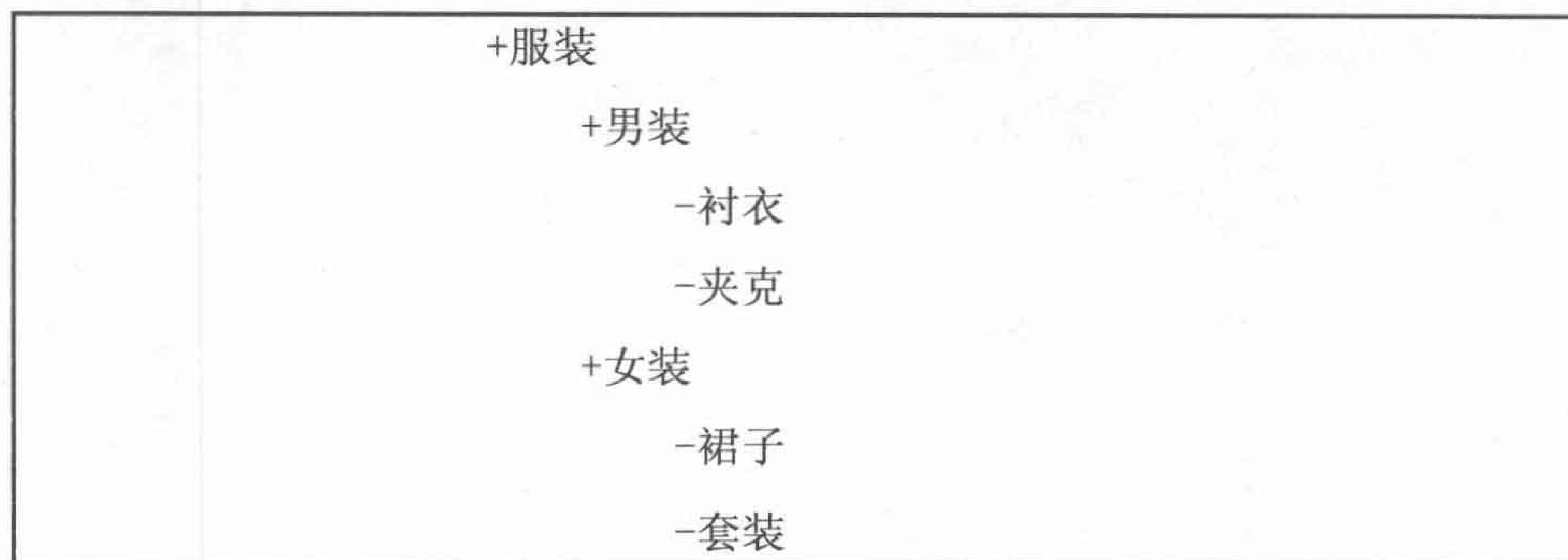
第 15 章　组合模式（Composite）

15.1 场 景 问 题

15.1.1 商品类别树

考虑这样一个实际的应用：管理商品类别树。

在实现跟商品有关的应用系统的时候，一个很常见的功能就是商品类别树的管理，比如有以下的商品类别树：



仔细观察上面的商品类别树，有以下几个明显的特点。

- 有一个根节点，比如服装，它没有父节点，它可以包含其他的节点。
- 树枝节点，有一类节点可以包含其他的节点，称之为树枝节点，比如男装、女装。
- 叶子节点，有一类节点没有子节点，称之为叶子节点，比如衬衣、夹克、裙子、套装。

现在需要管理商品类别树，假如要求能实现输出如上商品类别树的结构功能，应该如何实现呢？

15.1.2 不用模式的解决方案

要管理商品类别树，就是要管理树的各个节点。现在树上的节点有三类，根节点、树枝节点和叶子节点，再进一步分析发现，根节点和树枝节点是类似的，都是可以包含其他节点的节点，把它们称为容器节点。

这样一来，商品类别树的节点就被分成了两种，一种是容器节点，另一种是叶子节点。容器节点可以包含其他的容器节点或者叶子节点。把它们分别实现成为对象，也就是容器对象和叶子对象，容器对象可以包含其他的容器对象或者叶子对象。换句话说，容器对象是一种组合对象。

然后在组合对象和叶子对象里面去实现要求的功能就可以了，看看下面的代码实现。

（1）先来看看叶子对象的代码实现。示例代码如下：

```
/**
 * 叶子对象
 */
public class Leaf {
```

```
    /**
     * 叶子对象的名字
     */
    private String name = "";

    /**
     * 构造方法，传入叶子对象的名字
     * @param name 叶子对象的名字
     */
    public Leaf(String name){
        this.name = name;
    }

    /**
     * 输出叶子对象的结构，叶子对象没有子对象，也就是输出叶子对象的名字
     * @param preStr 前缀，主要是按照层级拼接的空格，实现向后缩进
     */
    public void printStruct(String preStr){
        System.out.println(preStr+"-"+name);
    }
}
```

（2）再来看看组合对象的代码实现。组合对象里面可以包含其他的组合对象或者是叶子对象，由于类型不一样，需要分开记录。示例代码如下：

```
/**
 * 组合对象，可以包含其他组合对象或者叶子对象
 */
public class Composite {
    /**
     * 用来记录包含的其他组合对象
     */
    private Collection<Composite> childComposite =
                                  new ArrayList<Composite>();
    /**
     * 用来记录包含的其他叶子对象
     */
    private Collection<Leaf> childLeaf = new ArrayList<Leaf>();
    /**
     * 组合对象的名字
     */
    private String name = "";
```

```
/**
 * 构造方法，传入组合对象的名字
 * @param name 组合对象的名字
 */
public Composite(String name){
    this.name = name;
}

/**
 * 向组合对象加入被它包含的其他组合对象
 * @param c 被它包含的其他组合对象
 */
public void addComposite(Composite c){
    this.childComposite.add(c);
}
/**
 * 向组合对象加入被它包含的叶子对象
 * @param leaf 被它包含的叶子对象
 */
public void addLeaf(Leaf leaf){
    this.childLeaf.add(leaf);
}
/**
 * 输出组合对象自身的结构
 * @param preStr 前缀，主要是按照层级拼接的空格，实现向后缩进
 */
public void printStruct(String preStr){
    //先把自己输出去
    System.out.println(preStr+"+"+this.name);
    //然后添加一个空格，表示向后缩进一个空格，输出自己包含的叶子对象
    preStr+=" ";
    for(Leaf leaf : childLeaf){
        leaf.printStruct(preStr);
    }
    //输出当前对象的子对象了
    for(Composite c : childComposite){
        //递归输出每个子对象
        c.printStruct(preStr);
    }
```

```
    }
}
```

（3）写个客户端来测试一下，看看是否能实现要求的功能。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //定义所有的组合对象
        Composite root = new Composite("服装");
        Composite c1 = new Composite("男装");
        Composite c2 = new Composite("女装");

        //定义所有的叶子对象
        Leaf leaf1 = new Leaf("衬衣");
        Leaf leaf2 = new Leaf("夹克");
        Leaf leaf3 = new Leaf("裙子");
        Leaf leaf4 = new Leaf("套装");

        //按照树的结构来组合组合对象和叶子对象
        root.addComposite(c1);
        root.addComposite(c2);
        c1.addLeaf(leaf1);
        c1.addLeaf(leaf2);
        c2.addLeaf(leaf3);
        c2.addLeaf(leaf4);

        //调用根对象的输出功能来输出整棵树
        root.printStruct("");
    }
}
```

运行一下，测试看看，是否能完成要求的功能。

15.1.3　有何问题

上面的实现，虽然能实现要求的功能，但是有一个很明显的问题：那就是必须区分组合对象和叶子对象，并进行有区别的对待，比如在 Composite 和 Client 里面，都需要去区别对待这两种对象。

区别对待组合对象和叶子对象，不仅让程序变得复杂，还对功能的扩展也带来不便。实际上，大多数情况下用户并不想要去区别它们，而是认为它们是一样的，这样他们操作起来最简单。

对于这种具有整体与部分关系，并能组合成树型结构的对象结构，如何才能够以一个统一的方式来进行操作呢？

15.2 解决方案

15.2.1 使用组合模式来解决问题

用来解决上述问题的一个合理的解决方案就是组合模式。那么什么是组合模式呢？

1. 组合模式的定义

将对象组合成树型结构以表示“部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

2. 应用组合模式来解决问题的思路

仔细分析上面不用模式的例子，要区分组合对象和叶子对象的根本原因，就在于没有把组合对象和叶子对象统一起来。也就是说，组合对象类型和叶子对象类型是完全不同的类型，这导致了操作的时候必须区分它们。

组合模式通过引入一个抽象的组件对象，作为组合对象和叶子对象的父对象，这样就把组合对象和叶子对象统一起来了，用户使用的时候，始终是在操作组件对象，而不再去区分是在操作组合对象还是叶子对象。

提示 **组合模式的关键就在于这个抽象类，这个抽象类既可以代表叶子对象，也可以代表组合对象，这样用户在操作的时候，对单个对象和组合对象的使用就具有了一致性。**

15.2.2 组合模式的结构和说明

组合模式的结构如图 15.1 所示。

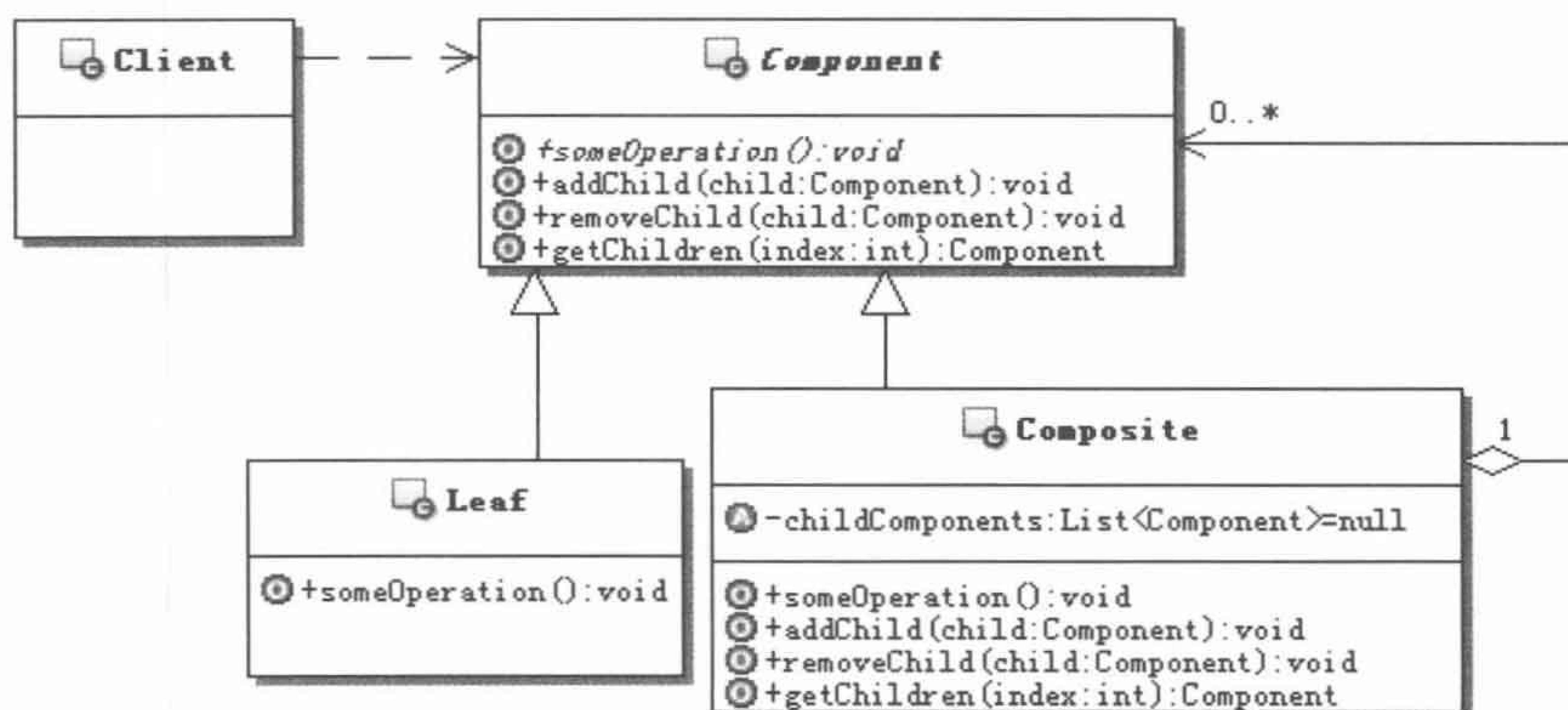


图 15.1 组合模式结构示意图

- Component：抽象的组件对象，为组合中的对象声明接口，让客户端可以通过这个接口来访问和管理整个对象结构，可以在里面为定义的功能提供缺省的实现。
- Leaf：叶子节点对象，定义和实现叶子对象的行为，不再包含其他的子节点对象。
- Composite：组合对象，通常会存储子组件，定义包含子组件的那些组件的行为，并实现在组件接口中定义的与子组件有关的操作。
- Client：客户端，通过组件接口来操作组合结构里面的组件对象。

一种典型的 Composite 对象结构通常是如图 15.2 所示的树型结构，一个 Composite 对象可以包含多个叶子对象和其他的 Composite 对象。虽然图 15.2 看起来好像有些对称，但那只是为了让图看起来美观一点，并不是说 Composite 组合的对象结构就是这样对称的，这点要提前说明一下。

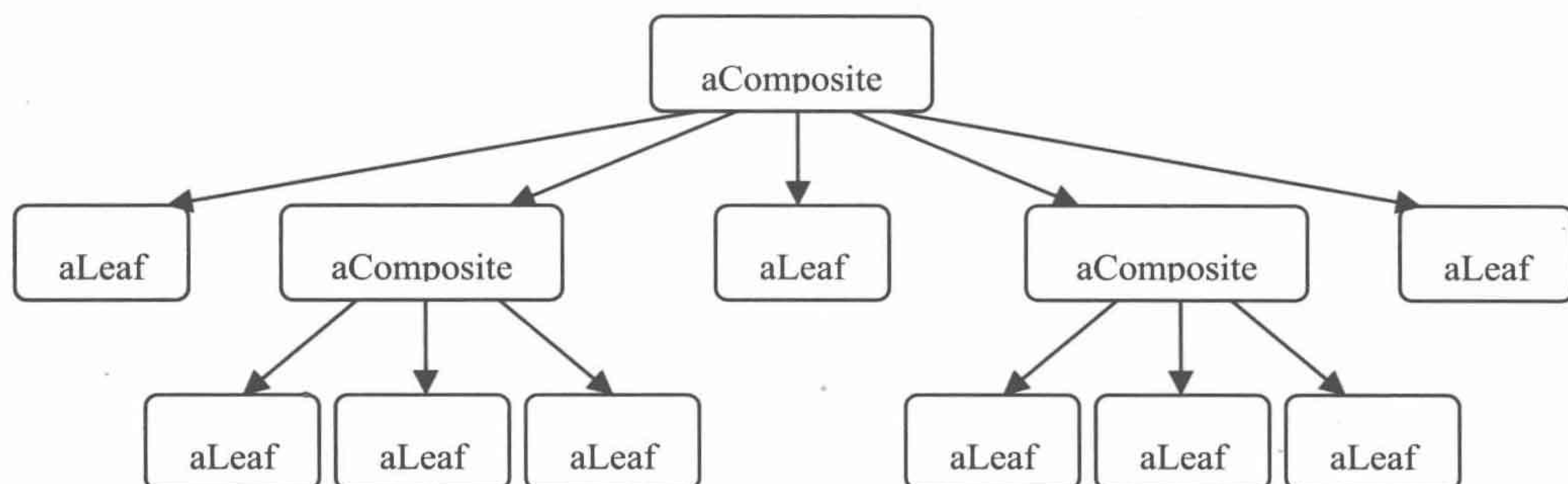


图 15.2　典型的 Composite 对象结构

15.2.3　组合模式示例代码

（1）先来看看组件对象的定义。示例代码如下：

```
/**
 * 抽象的组件对象，为组合中的对象声明接口，实现接口的缺省行为
 */
public abstract class Component {
    /**
     * 示意方法，子组件对象可能有的功能方法
     */
    public abstract void someOperation();
    /**
     * 向组合对象中加入组件对象
     * @param child 被加入组合对象中的组件对象
     */
    public void addChild(Component child) {
        // 缺省的实现，抛出例外，因为叶子对象没有这个功能
        //或者子组件没有实现这个功能
        throw new UnsupportedOperationException(
```

```
                                "对象不支持这个功能");
    }
    /**
     * 从组合对象中移出某个组件对象
     * @param child 被移出的组件对象
     */
    public void removeChild(Component child) {
        // 缺省的实现，抛出例外，因为叶子对象没有这个功能
        //或者子组件没有实现这个功能
        throw new UnsupportedOperationException(
                                "对象不支持这个功能");
    }
    /**
     * 返回某个索引对应的组件对象
     * @param index 需要获取的组件对象的索引，索引从0开始
     * @return 索引对应的组件对象
     */
    public Component getChildren(int index) {
        // 缺省的实现，抛出例外，因为叶子对象没有这个功能
        //或者子组件没有实现这个功能
        throw new UnsupportedOperationException(
                                "对象不支持这个功能");
    }
}
```

（2）接下来看看 Composite 对象的定义。示例代码如下：

```
/**
 * 组合对象，通常需要存储子对象，定义有子部件的部件行为
 * 并实现在Component里面定义的与子部件有关的操作
 */
public class Composite extends Component {
    /**
     * 用来存储组合对象中包含的子组件对象
     */
    private List<Component> childComponents = null;
    /**
     * 示意方法，通常在里面需要实现递归的调用
     */
    public void someOperation() {
        if (childComponents != null){
            for(Component c : childComponents){
```

```
                //递归地进行子组件相应方法的调用
                c.someOperation();
            }
        }
    }
    public void addChild(Component child) {
        //延迟初始化
        if (childComponents == null) {
            childComponents = new ArrayList<Component>();
        }
        childComponents.add(child);
    }
    public void removeChild(Component child) {
        if (childComponents != null) {
            childComponents.remove(child);
        }
    }
    public Component getChildren(int index) {
        if (childComponents != null){
            if(index>=0 && index<childComponents.size()){
                return childComponents.get(index);
            }
        }
        return null;
    }
}
```

（3）该来看看叶子对象的定义了。相对而言比较简单。示例代码如下：

```
/**
 * 叶子对象，叶子对象不再包含其他子对象
 */
public class Leaf extends Component {
    /**
     * 示意方法，叶子对象可能有自己的功能方法
     */
    public void someOperation() {
        // do something
    }
}
```

（4）对于 Client，就是使用 Component 接口来操作组合对象结构，由于使用方式千

差万别，这里仅仅提供一个示范性质的使用，顺便当作测试代码使用。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //定义多个Composite对象
        Component root = new Composite();
        Component c1 = new Composite();
        Component c2 = new Composite();
        //定义多个叶子对象
        Component leaf1 = new Leaf();
        Component leaf2 = new Leaf();
        Component leaf3 = new Leaf();

        //组合成为树形的对象结构
        root.addChild(c1);
        root.addChild(c2);
        root.addChild(leaf1);
        c1.addChild(leaf2);
        c2.addChild(leaf3);

        //操作Component对象
        Component o = root.getChildren(1);
        System.out.println(o);
    }
}
```

全部都是 Component 类型

并不是每次操作组件对象都需要组装树形的对象结构，如果对象结构已经存在，直接操作就可以了

15.2.4 使用组合模式重写示例

理解了组合模式的定义、结构和示例代码，对组合模式应该有一定的掌握了吧。下面就使用组合模式来重写前面不用模式的示例，看看用组合模式来实现会是什么样子，和不用模式有什么相同和不同之处。

为了整体理解和把握整个示例，先来看看示例的整体结构，如图 15.3 所示。

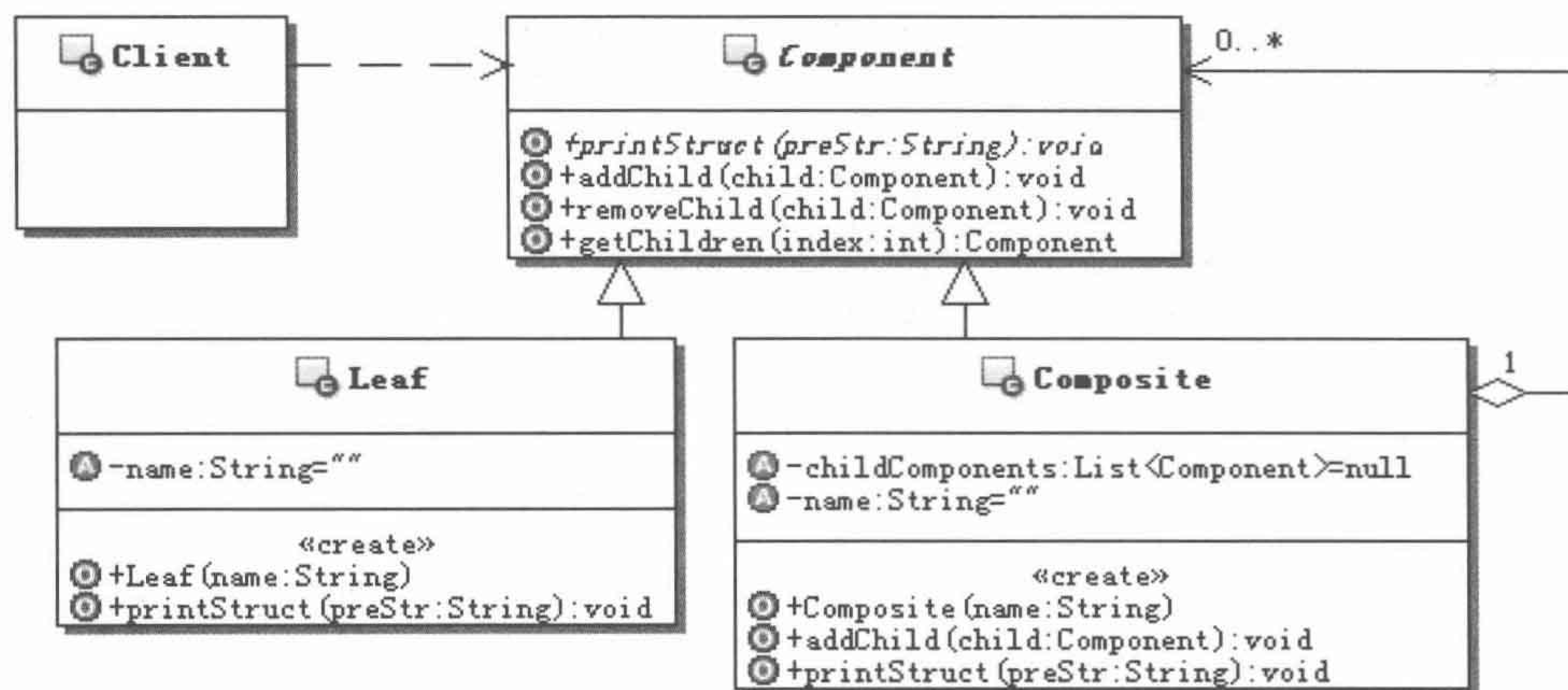


图 15.3　使用组合模式实现示例的结构示意图

（1）为组合对象和叶子对象添加一个抽象的父对象做为组件对象。在组件对象中，定义一个输出组件本身名称的方法以实现要求的功能。示例代码如下：

```
/**
 * 抽象的组件对象
 */
public abstract class Component {
    /**
     * 输出组件自身的名称
     */
    public abstract void printStruct(String preStr);
    /**
     * 向组合对象中加入组件对象
     * @param child 被加入组合对象中的组件对象
     */
    public void addChild(Component child) {
        throw new UnsupportedOperationException(
                                "对象不支持这个功能");
    }
    /**
     * 从组合对象中移出某个组件对象
     * @param child 被移出的组件对象
     */
    public void removeChild(Component child) {
        throw new UnsupportedOperationException(
                                "对象不支持这个功能");
    }
    /**
     * 返回某个索引对应的组件对象
```

相当于模式示例代码中的样例方法

没有改变，和模式示例代码一样

```
     * @param index 需要获取的组件对象的索引，索引从0开始
     * @return 索引对应的组件对象
     */
    public Component getChildren(int index) {
        throw new UnsupportedOperationException(
                                "对象不支持这个功能");
    }
}
```

（2）来看看叶子对象的实现，它的变化比较少，只是让叶子对象继承了组件对象，其他的和不用模式相比，没有什么变化。示例代码如下：

```
/**
 * 叶子对象
 */
public class Leaf extends Component{
    /**
     * 叶子对象的名字
     */
    private String name = "";
    /**
     * 构造方法，传入叶子对象的名字
     * @param name 叶子对象的名字
     */
    public Leaf(String name){
        this.name = name;
    }
    /**
     * 输出叶子对象的结构，叶子对象没有子对象，也就是输出叶子对象的名字
     * @param preStr 前缀，主要是按照层级拼接的空格，实现向后缩进
     */
    public void printStruct(String preStr){
        System.out.println(preStr+"-"+name);
    }
}
```


（3）接下来看看组合对象的实现，这个对象变化就比较多，大致有如下的改变。

- 新的 Composite 对象需要继承组件对象。
- 原来用来记录包含其他组合对象的集合和包含其他叶子对象的集合，被合并成为一个，就是统一的包含其他子组件对象的集合。使用组合模式来实现，不再需要区分到底是组合对象还是叶子对象了。
- 原来的 addComposite 和 addLeaf 方法，可以不需要了，将其合并实现成组件对象

中定义的 addChild 方法，但是需要现在的 Composite 来实现这个方法。使用组合模式来实现，不再需要区分到底是组合对象还是叶子对象了。

- 原来的 printStruct 方法的实现，完全要按照现在的方式来写，变化较大。

具体的示例代码如下：

```
/**
 * 组合对象，可以包含其他组合对象或者叶子对象
 */
public class Composite extends Component{
    /**
     * 用来存储组合对象中包含的子组件对象
     */
    private List<Component> childComponents = null;
    /**
     * 组合对象的名字
     */
    private String name = "";
    /**
     * 构造方法，传入组合对象的名字
     * @param name 组合对象的名字
     */
    public Composite(String name){
        this.name = name;
    }

    public void addChild(Component child) {
        //延迟初始化
        if (childComponents == null) {
            childComponents = new ArrayList<Component>();
        }
        childComponents.add(child);
    }
    /**
     * 输出组合对象自身的结构
     * @param preStr 前缀，主要是按照层级拼接的空格，实现向后缩进
     */
    public void printStruct(String preStr){
        //先把自己输出去
        System.out.println(preStr+"+"+this.name);
        //如果还包含有子组件，那么就输出这些子组件对象
        if(this.childComponents!=null){
```

需要继承组件对象

这一个集合，代替了原来分开记录的两个集合

代替了原来的 addComposite 和 addLeaf 方法

```
                //添加一个空格，表示向后缩进一个空格
                preStr+=" ";
                //输出当前对象的子对象
                for(Component c : childComponents){
                    //递归输出每个子对象
                    c.printStruct(preStr);
                }
            }
        }
    }
```


（4）客户端也有变化。客户端不再需要区分组合对象和叶子对象了，统一使用组件对象，调用的方法也都要改变成组件对象定义的方法。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //定义所有的组合对象
        Component root = new Composite("服装");
        Component c1 = new Composite("男装");
        Component c2 = new Composite("女装");

        //定义所有的叶子对象
        Component leaf1 = new Leaf("衬衣");
        Component leaf2 = new Leaf("夹克");
        Component leaf3 = new Leaf("裙子");
        Component leaf4 = new Leaf("套装");

        //按照树的结构来组合组合对象和叶子对象
        root.addChild(c1);
        root.addChild(c2);
        c1.addChild(leaf1);
        c1.addChild(leaf2);
        c2.addChild(leaf3);
        c2.addChild(leaf4);
        //调用根对象的输出功能来输出整棵树
        root.printStruct("");
    }
}
```

从上面的示例，大家可以看出，通过使用组合模式，把一个“部分—整体”的层次结构表示成了对象树的结构。这样一来，客户端就无需再区分操作的是组合对象还是叶子对象了；对于客户端而言，操作的都是组件对象。

15.3　模式讲解

15.3.1　认识组合模式

1. 组合模式的目的

组合模式的目的是：让客户端不再区分操作的是组合对象还是叶子对象，而是以一个统一的方式来操作。

实现这个目标的关键之处，是设计一个抽象的组件类，让它可以代表组合对象和叶子对象。这样一来，客户端就不用区分到底操作的是组合对象还是叶子对象了，只需要把它们全部当作组件对象进行统一的操作就可以了。

2. 对象树

通常，组合模式会组合出树型结构来，组成这个树型结构所使用的多个组件对象，就自然地形成了对象树。

这也意味着，所有可以使用对象树来描述或操作的功能，都可以考虑使用组合模式，比如读取 XML 文件，或是对语句进行语法解析等。

3. 组合模式中的递归

组合模式中的递归，指的是对象递归组合，不是常说的递归算法。通常我们谈的递归算法，是指“一个方法会调用方法自己”这样的算法，是从功能上来讲的，比如经典的求阶乘的例子。示例如下：

```
public class RecursiveTest {
    /**
     * 示意递归算法，求阶乘。这里只是简单的实现，只能实现求数值较小的阶乘
     * 对于数据比较大的阶乘，比如求100的阶乘应该采用java.math.BigDecimal
     * 或是java.math.BigInteger
     * @param a 求阶乘的数值
     * @return 该数值的阶乘值
     */
    public int recursive(int a){
        if(a==1){
            return 1;
        }
        return a * recursive(a-1);
    }
    public static void main(String[] args) {
        RecursiveTest test = new RecursiveTest();
        int result = test.recursive(5);
        System.out.println("5的阶乘="+result);
    }
```

```
}
```

而在组合模式中的递归，是对象本身的递归，是对象的组合方式，是从设计上来讲的，在设计上称作**递归关联**，是对象关联关系的一种。如果用 UML 来表示对象的递归关联的话，一对一的递归关联如图 15.4 所示，而一对多的递归关联如图 15.5 所示。

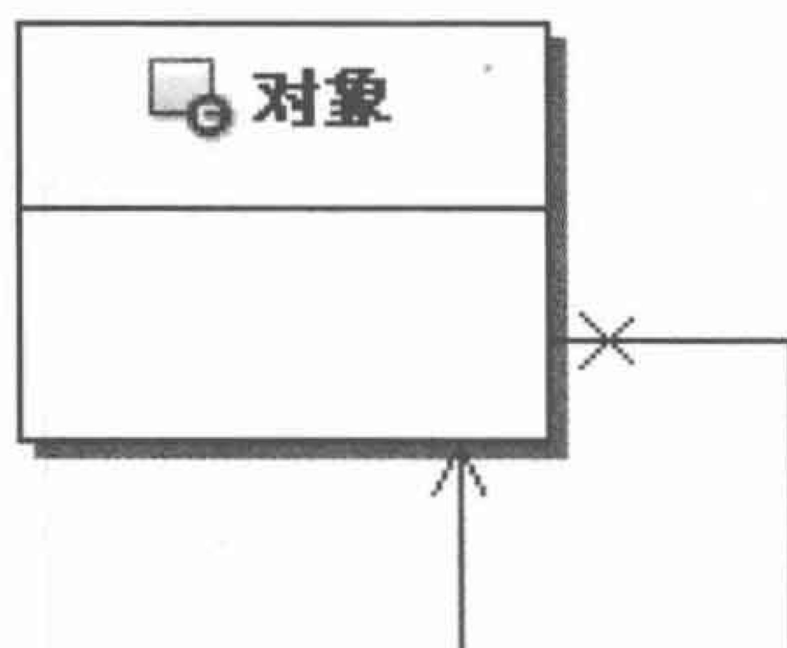


图 15.4 一对一递归关联结构示意图

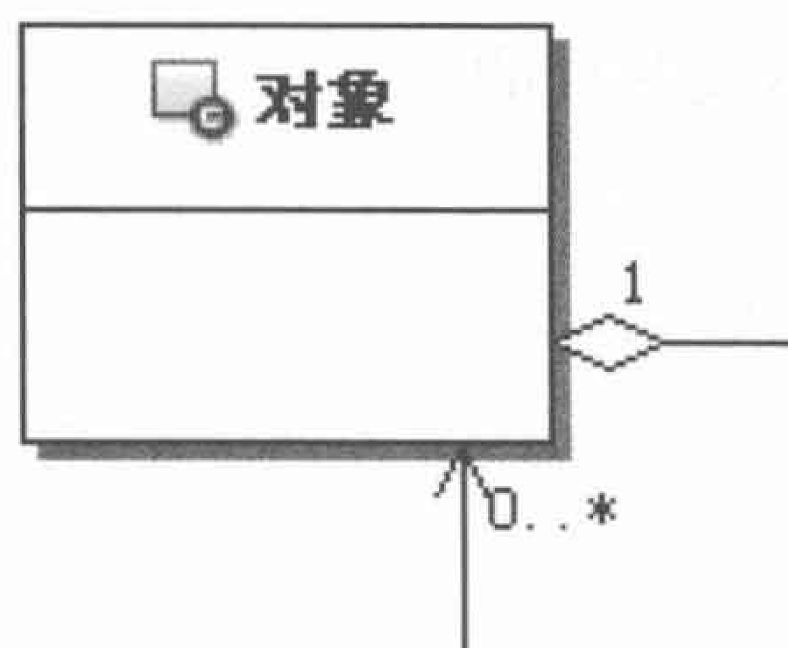


图 15.5 一对多递归关联结构示意图

另外，组合对象还有一个特点，就是理论上没有层次限制。组合对象 A 包含组合对象 B，组合对象 B 又包含组合对象 C…，这样下去是没有尽头的。因此在实现的时候，一个必然的选择就是递归实现。

4. Component 中是否应该实现一个 Component 列表

大多数情况下，一个 Composite 对象会持有子节点的集合。有些朋友可能就会想，那么能不能把这个子节点集合定义到 Component 中去呢？因为在 Component 中声明了一些操作子节点的方法，这样一来，大部分的工作就可以在 Component 中完成了。

事实上，这种方法是不太好的，因为在父类来存放子类的实例对象中，对于 Composite 节点来说没有什么，它本来就需要存放子节点；但是对于叶子节点来说，就会导致空间的浪费，因为叶节点本身不需要子节点。

因此只有当组合结构中叶子对象数目较少的时候，才值得使用这种方法。

5. 最大化 Component 定义

前面讲到了组合模式的目的是，让客户端不再区分操作的是组合对象还是叶子对象，而是以一种统一的方式来操作。

由于要统一两种对象的操作，所以 Component 中的方法也主要是两种对象对外方法的和。换句话说，有点大杂烩的意思，组件里面既有叶子对象需要的方法，也有组合对象需要的方法。

> 延伸 **其实这种实现是与类的设计原则相冲突的，类的设计有这样的原则：一个父类应该只定义那些对它的子类有意义的操作。但是看看上面的实现就知道，Component 中的有些方法对于叶子对象是没有意义的，那么怎么解决这一冲突呢？**

常见的做法是在 Component 中为对某些子对象没有意义的方法提供默认的实现，或是默认抛出不支持该功能的例外。这样一来，如果子对象需要这个功能，那就覆盖实现它，如果不需要，那就不用管了，使用父类的默认实现就可以了。

从另一个层面来说，如果把叶子对象看成是一个特殊的 Composite 对象，也就是没

有子节点的组合对象，这样，对于 Component 而言，子对象就被全部看做是组合对象，因此定义的所有方法都是有意义的了。

6. 子部件排序

在某些应用中，使用组合模式的时候，需要按照一定的顺序来使用子组件对象，比如进行语法分析的时候，使用组合模式构建的抽象语法树，在解析执行的时候，是需要按照顺序来执行的。

对于这样的功能，在设计的时候，需要把组件对象的索引考虑进去，并仔细地设计对子节点的访问和管理接口。通常的方式是需要按照顺序来存储，这样在获取的时候就可以按照顺序得到了。可以考虑结合 Iterator 模式来实现按照顺序来访问组件对象。

15.3.2　安全性和透明性

根据前面的讲述，在组合模式中，把组件对象分成了两种：一种是可以包含子组件的 Composite 对象；另一种是不能包含其他组件对象的叶子对象。

Composite 对象就像是一个容器，可以包含其他的 Composite 对象或叶子对象。当然有了容器，就要能对这个容器进行维护，需要向里面添加对象，并能够从容器里面获取对象，还要能从容器中删除对象，也就是说需要管理子组件对象。

> 提示　**这就产生了一个很重要的问题：在组合模式的类层次结构中，到底在哪一些类里面定义这些管理子组件的操作，是应该在 Component 中声明这些操作呢，还是在 Composite 中声明这些操作？**

这就需要仔细思考，在不同的实现中，进行安全性和透明性的权衡选择。

- 这里所说的安全性是指：从客户使用组合模式上看是否更安全。如果是安全的，那么就不会有发生误操作的可能，能访问的方法都是被支持的功能。
- 这里所说的透明性是指：从客户使用组合模式上，是否需要区分到底是组合对象还是叶子对象。如果是透明的，那就不用再区分，对于客户而言，都是组件对象，具体的类型对于客户而言是透明的，是客户无须关心的。

1. 透明性的实现

如果把管理子组件的操作定义在 Component 中，那么客户端只需要面对 Component，而无须关心具体的组件类型，这种实现方式就是透明性的实现。事实上，前面示例的实现方式都是这种实现方式。

但是透明性的实现是以安全性为代价的，因为在 Component 中定义的一些方法，对于叶子对象来说是没有意义的，比如增加、删除子组件对象。而客户不知道这些区别，对客户是透明的，因此客户可能会对叶子对象调用这种增加或删除子组件的方法，这样的操作是不安全的。

组合模式的透明性实现，通常的方式是：在 Component 中声明管理子组件的操作，并在 Component 中为这些方法提供默认的实现，如果子对象不支持的功能，默认的实现可以是抛出一个例外，来表示不支持这个功能。

2. 安全性的实现

如果把管理子组件的操作定义在 Composite 中，那么客户在使用叶子对象的时候，就不会发生使用添加子组件或是删除子组件的操作了，因为压根就没有这样的功能，这种实现方式是安全的。

但是这样一来，客户端在使用的时候，就必须区分到底使用的是 Composite 对象，还是叶子对象，不同对象的功能是不一样的。也就是说，这种实现方式，对客户而言就不是透明的了。

下面把用透明性方式实现的示例改成用安全性的方式再实现一次。这样大家可以对比来看，可以更好地理解组合模式的透明性和安全性这两种实现方式。

还是先来看一下使用安全性方式实现示例的结构，如图 15.6 所示。

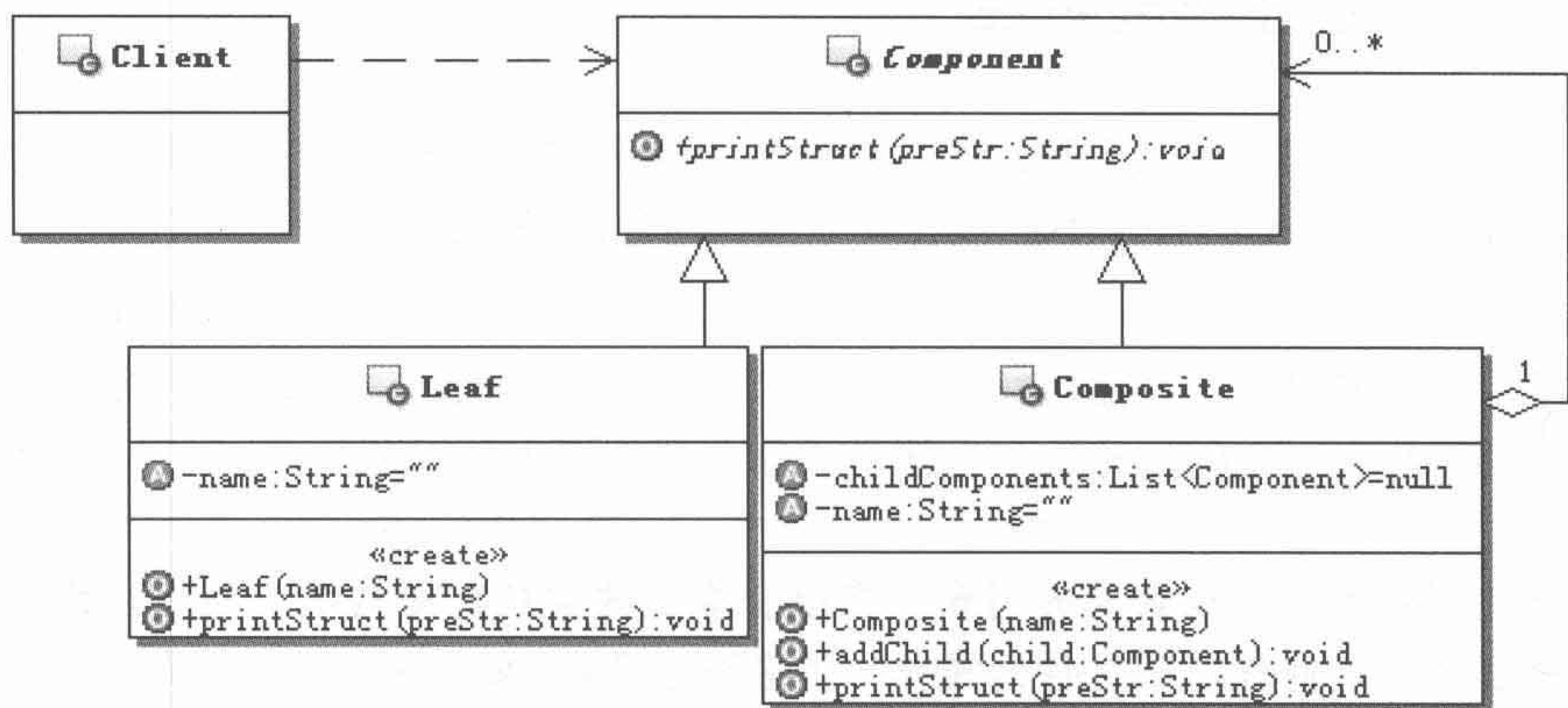


图 15.6 使用组合模式的安全性实现方式来实现示例的结构示意图

（1）首先看看 Component 的定义，跟透明性的实现相比，使用安全性的实现方式，Component 中不再定义管理和操作子组件的方法，把相应的方法都删除了。示例代码如下：

```
/**
 * 抽象的组件对象，安全性的实现方式
 */
public abstract class Component {
    /**
     * 输出组件自身的名称
     */
    public abstract void printStruct(String preStr);
}
```

（2）Composite 对象和 Leaf 对象的实现都没有任何的变化，这里就不再赘述。

（3）接下来看看客户端的实现。客户端的主要变化是要区分 Composite 对象和 Leaf 对象，而原来是不区分的，都是 Component 对象。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
```

```
        //定义所有的组合对象
        Composite root = new Composite("服装");
        Composite c1 = new Composite("男装");
        Composite c2 = new Composite("女装");
        //定义所有的叶子对象
        Leaf leaf1 = new Leaf("衬衣");
        Leaf leaf2 = new Leaf("夹克");
        Leaf leaf3 = new Leaf("裙子");
        Leaf leaf4 = new Leaf("套装");

        //按照树的结构来组合组合对象和叶子对象
        root.addChild(c1);
        root.addChild(c2);
        c1.addChild(leaf1);
        c1.addChild(leaf2);
        c2.addChild(leaf3);
        c2.addChild(leaf4);

        //调用根对象的输出功能来输出整棵树
        root.printStruct("");
    }
}
```

从上面的示例可以看出，从实现上，透明性和安全性的实现差别并不是很大。

3. 两种实现方式的选择

对于组合模式而言，在安全性和透明性上，会更看重透明性，毕竟组合模式的功能就是要让用户对叶子对象和组合对象的使用具有一致性。

而且对于安全性的实现，需要区分是组合对象还是叶子对象。有的时候，需要将对象进行类型转换，却发现类型信息丢失了，只好强行转换，这种类型转换必然是不够安全的。

对于这种情况的处理方法是在 Component 中定义一个 getComposite 方法，用来判断是组合对象还是叶子对象，如果是组合对象，就返回组合对象，如果是叶子对象，就返回 null，这样就实现了先判断，然后再强制转换。

因此在使用组合模式的时候，建议多采用透明性的实现方式，而少用安全性的实现方式。

15.3.3 父组件引用

在上面的示例中，都是在父组件对象中，保存有子组件的引用，也就是说都是从父到子的引用。而本节来讨论一下子组件对象到父组件对象的引用，它在实际开发中也是

非常有用的，比如：

- 现在要删除某个商品类别。如果这个类别没有子类别的话，直接删除就可以了，没有太大的问题，但是如果它还有子类别，这就涉及到它的子类别如何处理了，一种情况是连带全部删除，一种是上移一层，把被删除的商品类别对象的父商品类别，设置成为被删除的商品类别的子类别的父商品类别。
- 现在要进行商品类别的细化和调整，把原本属于 A 类别的一些商品类别，调整到 B 类别里面去，某个商品类别的调整会伴随着它所有的子类别一起调整。这样的调整可能会：把原本是兄弟关系的商品类别变成了父子关系，也可能会把原本是父子关系的商品类别调整成了兄弟关系，如此等等，会有很多种可能。

要实现上述的功能，一个较为简单的方案就是在保持从父组件到子组件引用的基础上，再增加保持从子组件到父组件的引用，这样在删除一个组件对象或是调整一个组件对象的时候，可以通过调整父组件的引用来实现，可以大大简化实现。

通常会在 Component 中定义对父组件的引用，组合对象和叶子对象都可以继承这个引用。那么什么时候来维护这个引用呢？

提示 **较为容易的办法就是：在组合对象添加子组件对象的时候，为子组件对象设置父组件的引用，在组合对象删除一个子组件对象的时候，再重新设置相关子组件的父组件引用。把这些实现到 Composite 中，这样所有的子类都可以继承到这些方法，从而更容易地维护子组件到父组件的引用。**

还是看示例会比较清楚。在前面实现的商品类别的示例基础上，来示例对父组件的引用，并实现删除某个商品类别，然后把被删除的商品类别对象的父商品类别，设置成为被删除的商品类别的子类别的父商品类别。也就是把被删除的商品类别对象的子商品类别都上移一层。

（1）先看看 Component 组件的定义，大致有如下变化。

- 添加一个属性来记录组件对象的父组件对象，同时提供相应的 getter/setter 方法来访问父组件对象。
- 添加一个能获取一个组件所包含的子组件对象的方法，提供给实现当某个组件被删除时，把它的子组件对象上移一层的功能时使用。

示例代码如下：

```
public abstract class Component {
    /**
     * 记录父组件对象
     */
    private Component parent = null;
    /**
     * 获取一个组件的父组件对象
     * @return 一个组件的父组件对象
     */
```

```
    public Component getParent() {
        return parent;
    }
    /**
     * 设置一个组件的父组件对象
     * @param parent 一个组件的父组件对象
     */
    public void setParent(Component parent) {
        this.parent = parent;
    }
    /**
     * 返回某个组件的子组件对象
     * @return 某个组件的子组件对象
     */
    public List<Component> getChildren() {
        throw new UnsupportedOperationException(
                                    "对象不支持这个功能");
    }
    /*--------------------以下是原有的定义----------------------*/
    public abstract void printStruct(String preStr);
    public void addChild(Component child) {
        throw new UnsupportedOperationException(
                                    "对象不支持这个功能");
    }
    public void removeChild(Component child) {
        throw new UnsupportedOperationException(
                                    "对象不支持这个功能");
    }
    public Component getChildren(int index) {
        throw new UnsupportedOperationException(
                                   "对象不支持这个功能");
    }
}
```

（2）接下来看看 Composite 的实现，大致有如下变化。

- 在添加子组件的方法实现中，加入对父组件的引用实现。
- 在删除子组件的方法实现中，加入把被删除的商品类别对象的父商品类别，设置成为被删除的商品类别的子类别的父商品类别的功能。
- 实现新的返回组件的子组件对象的功能。

示例代码如下：

```java
/**
 * 组合对象，可以包含其他组合对象或者叶子对象
 */
public class Composite extends Component{
    public void addChild(Component child) {
        //延迟初始化
        if (childComponents == null) {
            childComponents = new ArrayList<Component>();
        }
        childComponents.add(child);

        //添加对父组件的引用
        child.setParent(this);
    }

    public void removeChild(Component child) {
        if (childComponents != null) {
            //查找到要删除的组件在集合中的索引位置
            int idx = childComponents.indexOf(child);
            if (idx != -1) {
                //先把被删除的商品类别对象的父商品类别，
                //设置成为被删除的商品类别的子类别的父商品类别
                for(Component c : child.getChildren()){
                    //删除的组件对象是本实例的一个子组件对象
                    c.setParent(this);
                    //把被删除的商品类别对象的子组件对象添加到当前实例中
                    childComponents.add(c);
                }

                //真的删除
                childComponents.remove(idx);
            }
        }
    }
    public List<Component> getChildren() {
        return childComponents;
    }
    /*------------以下是原有的实现，没有变化----------------*/
    private List<Component> childComponents = null;
    private String name = "";
```

```
    public Composite(String name){
        this.name = name;
    }
    public void printStruct(String preStr){
        System.out.println(preStr+"+"+this.name);
        if(this.childComponents!=null){
            preStr+=" ";
            for(Component c : childComponents){
                c.printStruct(preStr);
            }
        }
    }
}
```

（3）叶子对象没有任何的改变，这里不再赘述。

（4）可以写个客户端来测试一下了。在原来的测试后面，删除一个节点，然后再次输出整棵树的结构，看看效果。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //定义所有的组合对象
        Component root = new Composite("服装");
        Component c1 = new Composite("男装");
        Component c2 = new Composite("女装");
        //定义所有的叶子对象
        Component leaf1 = new Leaf("衬衣");
        Component leaf2 = new Leaf("夹克");
        Component leaf3 = new Leaf("裙子");
        Component leaf4 = new Leaf("套装");
        //按照树的结构来组合组合对象和叶子对象
        root.addChild(c1);
        root.addChild(c2);
        c1.addChild(leaf1);
        c1.addChild(leaf2);
        c2.addChild(leaf3);
        c2.addChild(leaf4);
        //调用根对象的输出功能来输出整棵树
        root.printStruct("");
        System.out.println("------------------------------->");
        //然后删除一个节点
        root.removeChild(c1);
```

```
        //重新输出整棵树
        root.printStruct("");
    }
}
```

运行结果如下：

```
+服装
 +男装
  -衬衣
  -夹克
 +女装
  -裙子
  -套装
---------------------------->
+服装
 +女装
  -裙子
  -套装
 -衬衣
 -夹克
```

仔细观察上面的结果，当男装的节点被删除后，会把原来男装节点下的子节点，添加到原来男装的父节点，也就是服装的下面。输出是按照添加的先后顺序来的，所以先输出了女装节点，然后才输出衬衣和夹克节点。

15.3.4 环状引用

所谓环状引用，指的是在对象结构中，某个对象包含的子对象，或是子对象的子对象，或是子对象的子对象的子对象……如此经过 N 层后，出现所包含的子对象中有这个对象本身，从而构成了环状引用。比如，A 包含 B，B 包含 C，而 C 又包含了 A，转了一圈，转回来了，就构成了一个环状引用。

在使用组合模式构建树状结构的时候，这种引用是需要考虑的一种情况。通常情况下，组合模式构建的树状结构，是不应该出现环状引用的，如果出现了，多半是有错误发生了。因此在应用组合模式实现功能的时候，就应该考虑要检测并避免出现环状引用，否则很容易引起死循环，或是同一个功能被操作多次。

注意	**但是要说明的是：组合模式的实现里面也是可以有环状引用的，当然需要特殊构建环状引用，并提供相应的检测和处理，这里不去讨论这种情况。**

那么如何检测是否有环状引用的情况发生呢？

一个很简单的思路就是记录下每个组件从根节点开始的路径，因为要出现环状引用，在一条路径上，某个对象就必然会出现两次。因此每个对象在整个路径上只是出现了一

次，那么就不会出现环状引用。

这个判断的功能可以添加到 Composite 对象的添加子组件的方法中，如果是环状引用的话，就抛出例外，并不会把它加入到子组件中去。

还是通过示例来说明吧。在前面实现的商品类别的示例基础上，来加入对环状引用的检测和处理。约定用组件的名称来代表组件，也就是说，组件的名称是唯一的，不会重复的，只要检测在一条路径上，组件名称不重复，那么组件就不会重复。

（1）先看看 Component 的定义，大致有如下变化。

- 添加一个记录每个组件路径的属性，并提供相应的 getter/setter 方法。
- 为了拼接组件的路径，新添加一个方法来获取组件的名称。

示例代码如下：

```
public abstract class Component {
    /**
     * 记录每个组件的路径
     */
    private String componentPath = "";
    /**
     * 获取组件的路径
     * @return 组件的路径
     */
    public String getComponentPath() {
        return componentPath;
    }
    /**
     * 设置组件的路径
     * @param componentPath 组件的路径
     */
    public void setComponentPath(String componentPath) {
        this.componentPath = componentPath;
    }
    /**
     * 获取组件的名称
     * @return 组件的名称
     */
    public abstract String getName();
/*-------------------以下是原有的定义----------------------*/
    public abstract void printStruct(String preStr);
    public void addChild(Component child) {
        throw new UnsupportedOperationException(
                                         "对象不支持这个功能");
```

```
    }
    public void removeChild(Component child) {
        throw new UnsupportedOperationException(
                                    "对象不支持这个功能");
    }
    public Component getChildren(int index) {
        throw new UnsupportedOperationException(
                                    "对象不支持这个功能");
    }
}
```

（2）再看看 Composite 的实现，大致有如下变化。

- 提供获取组件名称的实现。
- 在添加子组件的实现方法中，进行是否环状引用的判断，并计算组件对象的路径，然后设置回组件对象中。

示例代码如下：

```
public class Composite extends Component{
    public String getName(){
        return this.name;
    }
    public void addChild(Component child) {
        //延迟初始化
        if (childComponents == null) {
            childComponents = new ArrayList<Component>();
        }
        childComponents.add(child);

        //先判断组件路径是否为空，如果为空，说明本组件是根组件
        if(this.getComponentPath()==null
                || this.getComponentPath().trim().length()==0){
            //把本组件的name设置到组件路径中
            this.setComponentPath(this.name);
        }
        //判断要加入的组件在路径上是否出现过
        //先判断是否是根组件
        if(this.getComponentPath()
                            .startsWith(child.getName()+".")){
            //说明是根组件，重复添加了
            throw new java.lang.IllegalArgumentException(
              "在本通路上，组件 '"+child.getName()+"' 已被添加过了");
```

```
        }else{
            if(this.getComponentPath()
                            .indexOf("."+child.getName()) < 0){
                //表示没有出现过，那么可以加入
                //计算组件的路径
                String componentPath = this.getComponentPath()
                            +"."+child.getName();
                //设置子组件的路径
                child.setComponentPath(componentPath);
            }else{
                throw new java.lang.IllegalArgumentException(
              "在本通路上，组件 '"+child.getName()+"' 已被添加过了");
            }
        }
    }
/*----------------以下是原有的实现，没有变化------------------*/
    private List<Component> childComponents = null;
    private String name = "";
    public Composite(String name){
        this.name = name;
    }
    public void printStruct(String preStr){
        System.out.println(preStr+"+"+this.name);
        if(this.childComponents!=null){
            preStr+=" ";
            for(Component c : childComponents){
                c.printStruct(preStr);
            }
        }
    }
}
```

（3）叶子对象的实现，只是多了一个实现获取组件名称的方法，也就是直接返回叶子对象的 Name，跟 Composite 中的实现是类似的，就不再代码示例了。

（4）客户端的代码可以不做修改，正常执行，输出商品类别树来。当然，如果想要看到环状引用检测的效果，则可以做一个环状引用测试看看，比如：

```
public class Client {
    public static void main(String[] args) {
        //定义所有的组合对象
        Component root = new Composite("服装");
```

```
        Component c1 = new Composite("男装");
        Component c2= new Composite("衬衣");
        Component c3= new Composite("男装");
        //设置一个环状引用
        root.addChild(c1);
        c1.addChild(c2);
        c2.addChild(c3);

        //调用根对象的输出功能来输出整棵树
        root.printStruct("");
    }
}
```

运行结果如下：

```
Exception in thread "main" java.lang.IllegalArgumentException:
    在本通路上，组件 '男装' 已被添加过了后面的堆栈信息就省略了
```

（5）说明。

上面进行环路检测的实现是非常简单的，但是还有一些问题没有考虑，比如，要是删除了路径上的某个组件对象，那么所有该组件对象的子组件对象所记录的路径，都需要修改，要把这个组件从所有相关路径上都删除。就是在被删除的组件对象的所有子组件对象的路径上，查找到被删除组件的名称，然后通过字符串截取的方式将其删除。

只是这样的实现方式有些不太好，要实现这样的功能，可以考虑使用动态计算路径的方式，每次添加一个组件的时候，动态地递归寻找父组件，然后父组件再找父组件，一直到根组件，这样就能避免某个组件被删除后，路径发生了变化，需要修改所有相关路径记录的情况。

15.3.5 组合模式的优缺点

组合模式有以下优点。

- 定义了包含基本对象和组合对象的类层次结构
  在组合模式中，基本对象可以被组合成复杂的组合对象，而组合对象又可以组合成更复杂的组合对象，可以不断地递归组合下去，从而构成一个统一的组合对象的类层次结构。
- 统一了组合对象和叶子对象
  在组合模式中，可以把叶子对象当作特殊的组合对象看待，为它们定义统一的父类，从而把组合对象和叶子对象的行为统一起来。
- 简化了客户端调用
  组合模式通过统一组合对象和叶子对象，使得客户端在使用它们的时候，不需要再去区分它们，客户不关心使用的到底是什么类型的对象，这就大大简化了客户端的使用。

- 更容易扩展

  由于客户端是统一地面对 Component 来操作，因此，新定义的 Composite 或 Leaf 子类能够很容易地与已有的结构一起工作，而客户端不需要为增添了新的组件类而改变。

组合模式的缺点是很难限制组合中的组件类型。

容易增加新的组件也会带来一些问题，比如很难限制组合中的组件类型。

这在需要检测组件类型的时候，使得我们不能依靠编译期的类型约束来完成，必须在运行期间动态检测。

15.3.6　思考组合模式

1. 组合模式的本质

> 组合模式的本质：**统一叶子对象和组合对象。**

组合模式通过把叶子对象当成特殊的组合对象看待，从而对叶子对象和组合对象一视同仁，全部当成了 Component 对象，有机地统一了叶子对象和组合对象。

正是因为统一了叶子对象和组合对象，在将对象构建成树型结构的时候，才不需要做区分，反正是组件对象里面包含其他的组件对象，如此递归下去；也才使得对于树形结构的操作变得简单，不管对象类型，统一操作。

2. 何时选用组合模式

建议在以下情况中选用组合模式。

- 如果你想表示对象的部分—整体层次结构，可以选用组合模式，把整体和部分的操作统一起来，使得层次结构实现更简单，从外部来使用这个层次结构也容易。
- 如果你希望统一地使用组合结构中的所有对象，可以选用组合模式，这正是组合模式提供的主要功能。

15.3.7　相关模式

- 组合模式和装饰模式

  这两个模式可以组合使用。

  装饰模式在组装多个装饰器对象的时候，是一个装饰器找下一个装饰器，下一个再找下一个，如此递归下去。其实这种结构也可以使用组合模式来帮助构建，这样一来，装饰器对象就相当于组合模式的 Composite 对象了。

  要让两个模式能很好地组合使用，通常会让它们有一个公共的父类。因此装饰器必须支持组合模式需要的一些功能，比如，增加、删除子组件等。
- 组合模式和享元模式

  这两个模式可以组合使用。

如果组合模式中出现大量相似的组件对象的话，可以考虑使用享元模式来帮助缓存组件对象，这样可以减少对内存的需要。

使用享元模式也是有条件的，如果组件对象的可变化部分的状态能够从组件对象中分离出去，并且组件对象本身不需要向父组件发送请求的话，就可以采用享元模式。

- 组合模式和迭代器模式

  这两个模式可以组合使用。

  在组合模式中，通常可以使用迭代器模式来遍历组合对象的子对象集合，而无需关心具体存放子对象的聚合结构。

- 组合模式和访问者模式

  这两个模式可以组合使用。

  访问者模式能够在不修改原有对象结构的情况下，为对象结构中的对象增添新的功能。访问者模式和组合模式合用，可以把原本分散在 Composite 和 Leaf 类中的操作和行为都局部化。

  如果在使用组合模式的时候，预计到今后可能会有增添其他功能的可能，那么可以采用访问者模式，来预留好添加新功能的方式和通道，这样以后在添加新功能的时候，就不需要再修改已有的对象结构和已经实现的功能了。

- 组合模式和职责链模式

  这两个模式可以组合使用。

  职责链模式要解决的问题是：实现请求的发送者和接收者之间解耦。职责链模式的实现方式是把多个接收者组合起来，构成职责链，然后让请求在这条链上传递，直到有接收者处理这个请求为止。

  可以应用组合模式来构建这条链，相当于是子组件找父组件，父组件又找父组件，如此递归下去，构成一条处理请求的组件对象链。

- 组合模式和命令模式

  这两个模式可以组合使用。

  命令模式中有一个宏命令的功能，通常这个宏命令就是使用组合模式来组装出来的。

第16章 模板方

（Template M

16.1 场 景 问 题

16.1.1 登录控制

几乎所有的应用系统，都需要系统登录控制的功能，有些系统甚至有多个登录控制的功能，比如，普通用户可以登录前台，进行相应的业务操作；而工作人员可以登录后台，进行相应的系统管理或业务处理。

现在有这么一个基于 Web 的企业级应用系统，需要实现这两种登录控制，直接使用不同的登录页面来区分它们，把基本的功能需求分别描述如下。

先看看普通用户登录前台的登录控制的功能。

- 前台页面：用户能输入用户名和密码；提交登录请求，让系统进行登录控制。
- 后台：从数据库获取登录人员的信息。
- 后台：判断从前台传递过来的登录数据和数据库中已有的数据是否匹配。
- 前台 Action：如果匹配就转向首页，如果不匹配就返回到登录页面，并显示错误提示信息。

再来看看工作人员登录后台的登录控制功能。

- 前台页面：用户能输入用户名和密码；提交登录请求，让系统进行登录控制。
- 后台：从数据库获取登录人员的信息。
- 后台：把从前台传递过来的密码数据使用相应的加密算法进行加密运算，得到加密后的密码数据。
- 后台：判断从前台传递过来的用户名和加密后的密码数据和数据库中已有的数据是否匹配。
- 前台 Action：如果匹配就转向首页，如果不匹配就返回到登录页面，并显示错误提示信息。

> 提示　**说明：普通用户和工作人员在数据库中是存储在不同表里面的，当然也是不同的模块来维护普通用户的数据和工作人员的数据，另外工作人员的密码是加密存放的。**

16.1.2 不用模式的解决方案

由于普通用户登录和工作人员登录是不同的模块，有不同的页面、不同的逻辑处理和不同的数据存储，因此，在实现上完全作为两个独立的小模块来完成。这里把它们的逻辑处理部分分别实现出来。

（1）先来看看普通用户登录的逻辑处理部分。示例代码如下：

```
/**
 * 普通用户登录控制的逻辑处理
```

```
 */
public class NormalLogin {
    /**
     * 判断登录数据是否正确，也就是是否能登录成功
     * @param lm 封装登录数据的Model
     * @return true表示登录成功，false表示登录失败
     */
    public boolean login(LoginModel lm) {
        //1：从数据库获取登录人员的信息，就是根据用户编号去获取人员的数据
        UserModel um = this.findUserByUserId(lm.getUserId());
        //2：判断从前台传递过来的登录数据和数据库中已有的数据是否匹配
        //先判断用户是否存在，如果um为null，说明用户肯定不存在
        //但是不为null，用户不一定存在，因为数据层可能返回new UserModel();
        //因此还需要做进一步的判断
        if (um != null) {
            //如果用户存在，检查用户编号和密码是否匹配
            if (um.getUserId().equals(lm.getUserId())
                    && um.getPwd().equals(lm.getPwd())) {
                return true;
            }
        }
        return false;
    }
    /**
     * 根据用户编号获取用户的详细信息
     * @param userId 用户编号
     * @return 对应的用户的详细信息
     */
    private UserModel findUserByUserId(String userId) {
        // 这里省略具体的处理，仅做示意，返回一个有默认数据的对象
        UserModel um = new UserModel();
        um.setUserId(userId);
        um.setName("test");
        um.setPwd("test");
        um.setUuid("User0001");
        return um;
    }
}
```

对应的 LoginModel，示例代码如下：

```
/**
 * 描述登录人员登录时填写的信息的数据模型
 */
public class LoginModel {
    private String userId,pwd;
    public String getUserId() {
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

对应的 UserModel，示例代码如下：

```
/**
 * 描述用户信息的数据模型
 */
public class UserModel {
    private String uuid,userId,pwd,name;
    public String getUuid() {
        return uuid;
    }
    public void setUuid(String uuid) {
        this.uuid = uuid;
    }
    public String getUserId() {
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
```

```
        this.pwd = pwd;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

（2）再看看工作人员登录的逻辑处理部分。示例代码如下：

```
/**
 * 工作人员登录控制的逻辑处理
 */
public class WorkerLogin {
    /**
     * 判断登录数据是否正确，也就是是否能登录成功
     * @param lm 封装登录数据的Model
     * @return true表示登录成功，false表示登录失败
     */
    public boolean login(LoginModel lm) {
        //1：根据工作人员编号去获取工作人员的数据
        WorkerModel wm = findWorkerByWorkerId(lm.getWorkerId());
        //2：判断从前台传递过来的用户名和加密后的密码数据，
       //和数据库中已有的数据是否匹配
        //先判断工作人员是否存在，如果wm为null，说明工作人员肯定不存在
        //但是不为null，工作人员不一定存在，
        //因为数据层可能返回new WorkerModel();因此还需要做进一步的判断
        if (wm != null) {
            //3：把从前台传来的密码数据使用相应的加密算法进行加密运算
            String encryptPwd = this.encryptPwd(lm.getPwd());
            //如果工作人员存在，检查工作人员编号和密码是否匹配
            if (wm.getWorkerId().equals(lm.getWorkerId())
                    && wm.getPwd().equals(encryptPwd)) {
                return true;
            }
        }
        return false;
    }
    /**
     * 对密码数据进行加密
```

注意这个地方，是和加密后的密码数据进行比较

```
     * @param pwd 密码数据
     * @return 加密后的密码数据
     */
    private String encryptPwd(String pwd){
        //这里对密码进行加密，省略了
        return pwd;
    }
    /**
     * 根据工作人员编号获取工作人员的详细信息
     * @param workerId 工作人员编号
     * @return 对应的工作人员的详细信息
     */
    private WorkerModel findWorkerByWorkerId(String workerId) {
        // 这里省略具体的处理，仅做示意，返回一个有默认数据的对象
        WorkerModel wm = new WorkerModel();
        wm.setWorkerId(workerId);
        wm.setName("Worker1");
        wm.setPwd("worker1");
        wm.setUuid("Worker0001");
        return wm;
    }
}
```

对应的 LoginModel，示例代码如下：

```
/**
 * 描述登录人员登录时填写的信息的数据模型
 */
public class LoginModel{
    private String workerId,pwd;
    public String getWorkerId() {
        return workerId;
    }
    public void setWorkerId(String workerId) {
        this.workerId = workerId;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
```

不是 userId，而是 workerId 了

就是对应的 getter/setter

```
        this.pwd = pwd;
    }
}
```

对应的 WorkerModel，示例代码如下：

```
/**
 * 描述工作人员信息的数据模型
 */
public class WorkerModel {
    private String uuid,workerId,pwd,name;
    public String getUuid() {
        return uuid;
    }
    public void setUuid(String uuid) {
        this.uuid = uuid;
    }
    public String getWorkerId() {
        return workerId;
    }
    public void setWorkerId(String workerId) {
        this.workerId = workerId;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

不是 userId，而是 workerId 了

就是对应的 getter/setter

16.1.3 有何问题

看了上面的实现示例，是不是很简单。但是，仔细看看，总会觉得有点问题，两种登录的实现太相似了，现在是完全分开，当作两个独立的模块来实现的，如果今后要扩展功能，比如要添加“控制同一个编号同时只能登录一次”的功能，那么两个模块都需要修改，是很麻烦的。而且，现在的实现中，也有很多相似的地方，显得很重复；另外，具体的实现和判断的步骤混合在一起，不利于今后变换功能，比如要变换加密算法等。

总之，上面的实现，有两个很明显的问题：一是重复或相似代码太多；二是扩展起来很不方便。

那么该怎样解决呢？如何实现才能让系统既灵活又能简洁地实现需求功能呢？

16.2 解决方案

16.2.1 使用模板方法模式来解决问题

用来解决上述问题的一个合理的解决方案就是模板方法模式。那么什么是模板方法模式呢？

1. 模板方法模式的定义

> 定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

2. 应用模板方法模式来解决问题的思路

仔细分析上面的问题，重复或相似代码太多、扩展不方便，出现这些问题的原因在哪里？主要就是两个实现是完全分开、相互独立的，没有从整体上进行控制。如果把两个模块合起来看，就会发现，那些重复或相似的代码应该被抽取出来，做成公共的功能，而不同的登录控制就可以去扩展这些公共的功能。这样一来，扩展的时候，如果出现有相同的功能，直接扩展公共功能就可以了。

使用模板方法模式，就可以很好地实现上面的思路。分析上面两个登录控制模块，会发现它们在实现上有着大致相同的步骤，只是在每步具体的实现上，略微有些不同。因此，可以把这些运算步骤看做是算法的骨架，把具体的不同的步骤实现延迟到子类去实现，这样就可以通过子类来提供不同的功能实现了。

经过分析总结，登录控制大致的逻辑判断步骤如下。

（1）根据登录人员的编号去获取相应的数据。

（2）获取对登录人员填写的密码数据进行加密后的数据，如果不需要加密，那就直接返回登录人员填写的密码数据。

（3）判断登录人员填写的数据和从数据库中获取的数据是否匹配。

在这三个步骤里面，第一个和第三个步骤是必不可少的，而第二个步骤是可选的。那么就可以定义一个父类，在其中定义一个方法来定义这个算法骨架，这个方法就是模板方法，然后把父类无法确定的实现，延迟到具体的子类来实现就可以了。

通过这样的方式，如果要修改加密的算法，那就在模板的子类里面重新覆盖实现加密的方法就可以了，完全不需要去改变父类的算法结构，即可重新定义这些特定的步骤。

16.2.2　模板方法模式的结构和说明

模板方法模式的结构如图 16.1 所示。

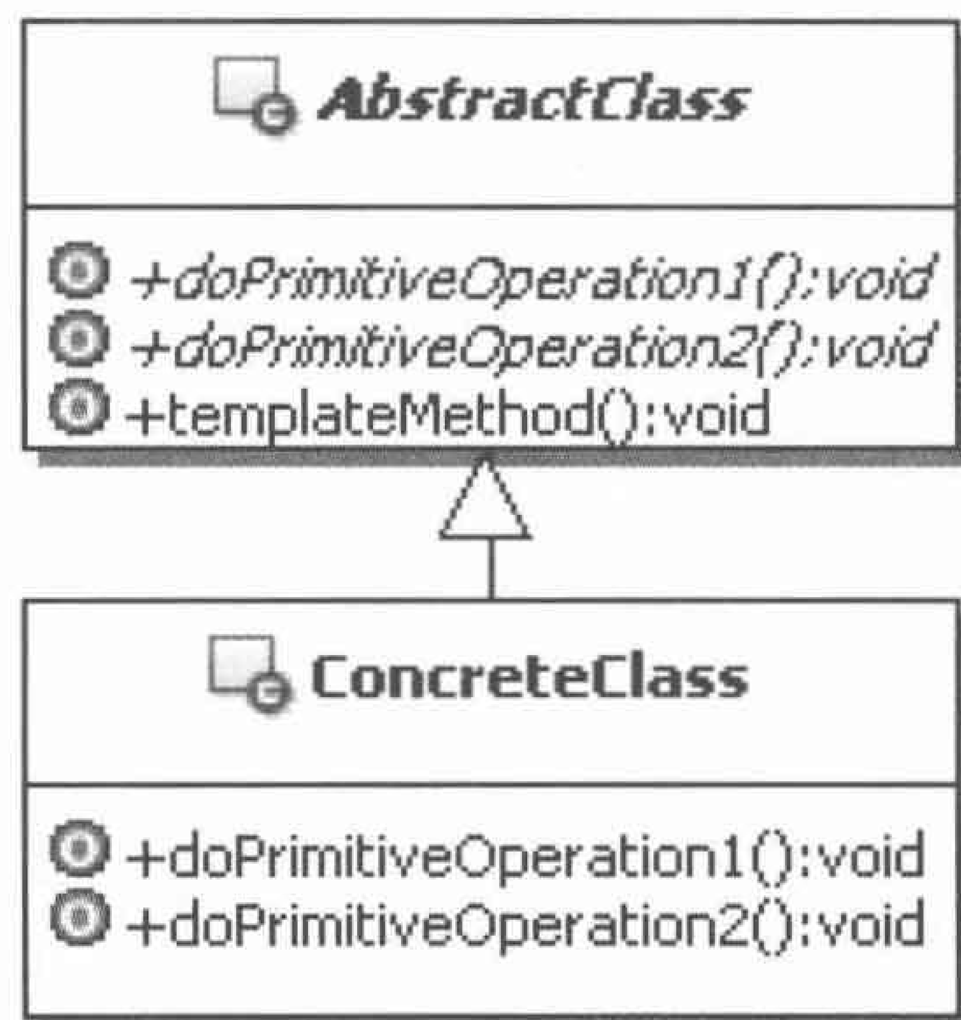


图 16.1　模板方法模式的结构示意图

- AbstractClass：抽象类。用来定义算法骨架和原语操作，具体的子类通过重定义这些原语操作来实现一个算法的各个步骤。在这个类里面，还可以提供算法中通用的实现。
- ConcreteClass：具体实现类。用来实现算法骨架中的某些步骤，完成与特定子类相关的功能。

16.2.3　模板方法模式示例代码

（1）先来看看 AbstractClass 的写法。示例代码如下：

```
/**
 * 定义模板方法、原语操作等的抽象类
 */
public abstract class AbstractClass {
    /**
     * 原语操作1，所谓原语操作就是抽象的操作，必须要由子类提供实现
     */
    public abstract void doPrimitiveOperation1();
    /**
     * 原语操作2
     */
```

```
    public abstract void doPrimitiveOperation2();
    /**
     * 模板方法，定义算法骨架
     */
    public final void templateMethod() {
        doPrimitiveOperation1();
        doPrimitiveOperation2();
    }
}
```

（2）再看看具体实现类的写法。示例代码如下：

```
/**
 * 具体实现类，实现原语操作
 */
public class ConcreteClass extends AbstractClass {
    public void doPrimitiveOperation1() {
        //具体的实现
    }
    public void doPrimitiveOperation2() {
        //具体的实现
    }
}
```

16.2.4 使用模板方法模式重写示例

要使用模板方法模式来实现前面的示例，需要按照模板方法模式的定义和结构，定义出一个抽象的父类，在这个父类中定义模板方法，这个模板方法应该实现进行登录控制的整体的算法步骤。对于公共的功能，就放到这个父类中实现，而这个父类无法决定的功能，就延迟到子类去实现。

这样一来，两种登录控制就做为这个父类的子类，分别实现自己需要的功能。此时系统的结构如图 16.2 所示。

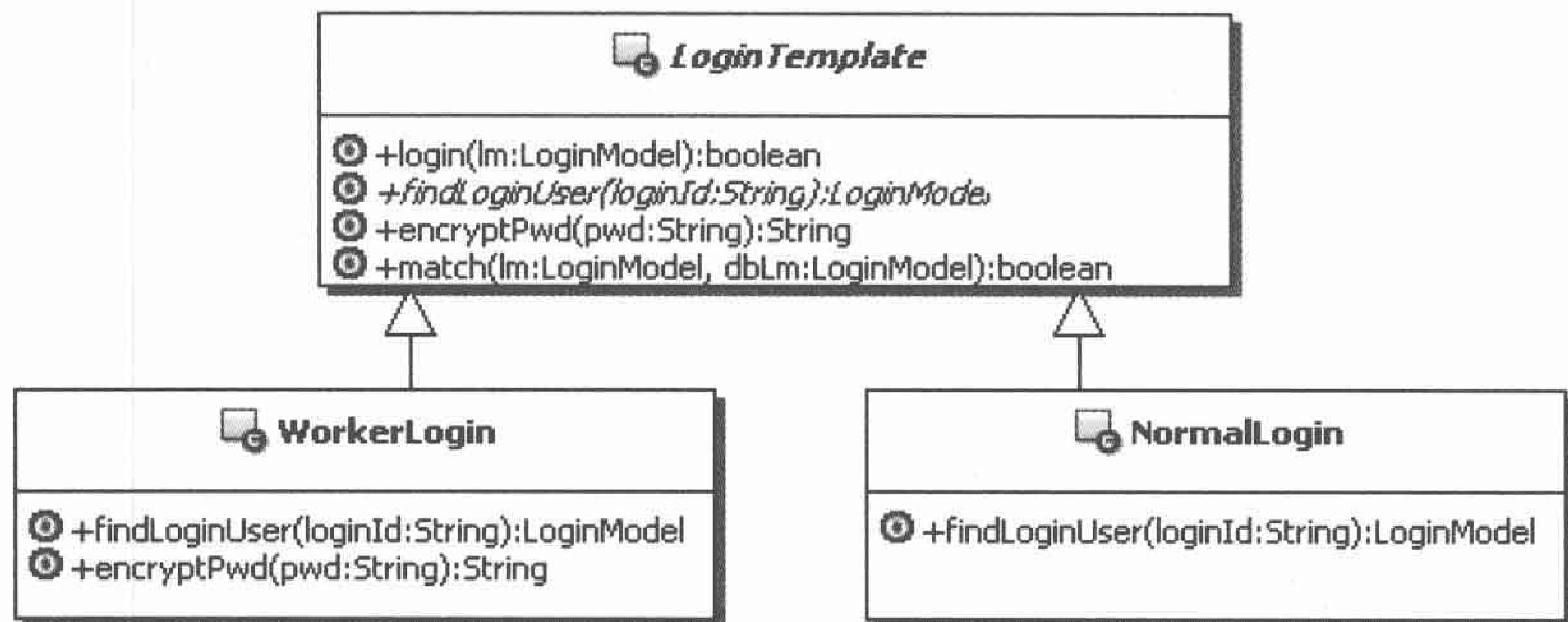


图 16.2　使用模板方法模式实现示例的结构示意图

（1）为了把原来的两种登录控制统一起来，首先需要把封装登录控制所需要的数据模型统一起来，不再区分是用户编号还是工作人员编号，而统一称为登录人员编号，并且将其他用不上的数据删除，这样直接使用一个数据模型就可以了。当然，如果各个子类实现需要其他的数据，还可以自行扩展。示例代码如下：

```
/**
 * 封装进行登录控制所需要的数据
 */
public class LoginModel {
    /**
     * 登录人员的编号，通用的，可能是用户编号，也可能是工作人员编号
     */
    private String loginId;
    /**
     * 登录的密码
     */
    private String pwd;
    public String getLoginId() {
        return loginId;
    }
    public void setLoginId(String loginId) {
        this.loginId = loginId;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

（2）接下来定义公共的登录控制算法骨架。示例代码如下：

```
/**
 *  登录控制的模板
 */
public abstract class LoginTemplate {
    /**
     * 判断登录数据是否正确，也就是是否能登录成功
     * @param lm 封装登录数据的Model
     * @return true表示登录成功，false表示登录失败
     */
```

```
public final boolean login(LoginModel lm){
    //1：根据登录人员的编号去获取相应的数据
    LoginModel dbLm = this.findLoginUser(lm.getLoginId());
    if(dbLm!=null){
        //2：对密码进行加密
        String encryptPwd = this.encryptPwd(lm.getPwd());
        //把加密后的密码设置回到登录数据模型中
        lm.setPwd(encryptPwd);
        //3：判断是否匹配
        return this.match(lm, dbLm);
    }
    return false;
}
/**
 * 根据登录编号来查找和获取存储中相应的数据
 * @param loginId 登录编号
 * @return 登录编号在存储中相对应的数据
 */
public abstract LoginModel findLoginUser(String loginId);
/**
 * 对密码数据进行加密
 * @param pwd 密码数据
 * @return 加密后的密码数据
 */
public String encryptPwd(String pwd){
    return pwd;
}
/**
 * 判断用户填写的登录数据和存储中对应的数据是否匹配得上
 * @param lm 用户填写的登录数据
 * @param dbLm 在存储中对应的数据
 * @return true表示匹配成功，false表示匹配失败
 */
public boolean match(LoginModel lm,LoginModel dbLm){
    if(lm.getLoginId().equals(dbLm.getLoginId())
            && lm.getPwd().equals(dbLm.getPwd())){
        return true;
    }
    return false;
}
```

```
}
```

（3）实现新的普通用户登录控制的逻辑处理。示例代码如下：

```
/**
 * 普通用户登录控制的逻辑处理
 */
public class NormalLogin extends LoginTemplate{
    public LoginModel findLoginUser(String loginId) {
        // 这里省略具体的处理，仅做示意，返回一个有默认数据的对象
        LoginModel lm = new LoginModel();
        lm.setLoginId(loginId);
        lm.setPwd("testpwd");
        return lm;
    }
}
```

（4）实现新的工作人员登录控制的逻辑处理。示例代码如下：

```
/**
 * 工作人员登录控制的逻辑处理
 */
public class WorkerLogin extends LoginTemplate{
    public LoginModel findLoginUser(String loginId) {
        // 这里省略具体的处理，仅做示意，返回一个有默认数据的对象
        LoginModel lm = new LoginModel();
        lm.setLoginId(loginId);
        lm.setPwd("workerpwd");
        return lm;
    }

    public String encryptPwd(String pwd){
        //覆盖父类的方法，提供真正的加密实现
        //这里对密码进行加密，比如使用MD5、3DES等，省略了
        System.out.println("使用MD5进行密码加密");
        return pwd;
    }
}
```

通过上面的示例，可以看出来，把原来的实现改成使用模板方法模式来实现也并不困难。写个客户端测试一下，以便更好地体会。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备登录人的信息
```

```
        LoginModel lm = new LoginModel();
        lm.setLoginId("admin");
        lm.setPwd("workerpwd");

        //准备用来进行判断的对象
        LoginTemplate lt = new WorkerLogin();
        LoginTemplate lt2 = new NormalLogin();

        //进行登录测试
        boolean flag = lt.login(lm);
        System.out.println("可以登录工作平台="+flag);

        boolean flag2 = lt2.login(lm);
        System.out.println("可以进行普通人员登录="+flag2);
    }
}
```

运行结果示例如下：

```
使用MD5进行密码加密
可以登录工作平台=true
可以进行普通人员登录=false
```

当然，你可以使用不同的测试数据来测试这个示例。

16.3 模式讲解

16.3.1 认识模板方法模式

1. 模板方法模式的功能

模板方法模式的功能在于固定算法骨架，而让具体算法实现可扩展。

这在实际应用中非常广泛，尤其是在设计框架级功能的时候非常有用。框架定义好了算法的步骤，在合适的点让开发人员进行扩展，实现具体的算法。比如在DAO实现中设计通用的增删改查功能，这个在后面会给大家示例。

模板方法模式还额外提供了一个好处，就是可以控制子类的扩展。因为在父类中定义好了算法的步骤，只是在某几个固定的点才会调用到被子类实现的方法，因此也就只允许在这几个点来扩展功能。这些可以被子类覆盖以扩展功能的方法通常被称为“钩子”方法，在后面也会给大家示例。

2. 为何不是接口

有的朋友可能会问一个问题，不是说在Java中应该尽量面向接口编程吗，为何模板方法的模板采用的是抽象方法呢？

要回答这个问题，首先搞清楚抽象类和接口的关系：

- 接口是一种特殊的抽象类，所有接口中的属性自动是常量，也就是 public final static 的，而所有接口中的方法必须是抽象的。
- 抽象类，简单点说是用 abstract 修饰的类。这里要特别注意的是抽象类和抽象方法的关系，记住两句话：**抽象类不一定包含抽象方法；有抽象方法的类一定是抽象类。**
- 抽象类和接口相比较，最大的特点就在于抽象类中是可以有具体的实现方法的，而接口中所有的方法都是没有具体的实现的。

> **延伸** 因此，虽然 Java 编程中倡导大家“面向接口编程”，并不是说就不再使用抽象类了。那么什么时候使用抽象类呢？
> 通常在**“既要约束子类的行为，又要为子类提供公共功能”的时候使用抽象类。**

按照这个原则来思考模板方法模式的实现，模板方法模式需要固定定义算法的骨架，这个骨架应该只有一份，算是一个公共的行为，但其中具体的步骤的实现又可能是各不相同的，恰好符合选择抽象类的原则。

把模板实现成为抽象类，为所有的子类提供了公共的功能，就是定义了具体的算法骨架；同时在模板中把需要由子类扩展的具体步骤的算法定义成为抽象方法，要求子类去实现这些方法，这就约束了子类的行为。

因此综合考虑，用抽象类来实现模板是一个很好的选择。

3. 变与不变

程序设计的一个很重要的思考点就是“变与不变”，也就是分析程序中哪些功能是可变的，哪些功能是不变的，然后把不变的部分抽象出来，进行公共的实现，把变化的部分分离出去，用接口来封装隔离，或者是用抽象类来约束子类行为。

模板方法模式很好地体现了这一点。模板类实现的就是不变的方法和算法的骨架，而需要变化的地方，都通过抽象方法，把具体实现延迟到子类中了，而且还通过父类的定义来约束了子类的行为，从而使系统能有更好的复用性和扩展性。

4. 好莱坞法则

什么是好莱坞法则呢？简单点说，就是“不要找我们，我们会联系你”。

模板方法模式很好地体现了这一点，作为父类的模板会在需要的时候，调用子类相应的方法，也就是由父类来找子类，而不是让子类来找父类。

这其实也是一种反向的控制结构。按照通常的思路，是子类找父类才对，也就是应该是子类来调用父类的方法，因为父类根本就不知道子类，而子类是知道父类的，但是在模板方法模式里面，是父类来找子类，所以是一种反向的控制结构。

那么，在 Java 里面能实现这样功能的理论依据在哪里呢？

理论依据就在于 Java 的动态绑定采用的是“后期绑定”技术，对于出现子类覆盖父

类方法的情况，在编译时是看数据类型，运行时则看实际的对象类型（new 操作符后跟的构造方法是哪个类的）。**一句话：new 谁就调用谁的方法**。

因此在使用模板方法模式的时候，虽然用的数据类型是模板类型，但是在创建类实例的时候是创建的具体的子类的实例，在调用的时候，会被动态绑定到子类的方法上，从而实现反向控制。其实在写父类的时候，它调用的方法是父类自己的抽象方法，只是在运行的时候被动态绑定到了子类的方法上。

5. 扩展登录控制

在使用模板方法模式实现以后，如果想要扩展新的功能，有以下几种情况。

一种情况是只需要提供新的子类实现就可以了。比如想要切换不同的加密算法，现在使用的是 MD5，如果想要实现使用 3DES 的加密算法，那就新做一个子类，然后覆盖实现父类加密的方法，在里面使用 3DES 来实现即可，已有的实现不需要做任何变化。

另外一种情况是想要给两个登录模块都扩展同一个功能，这种情况多属于需要修改模板方法的算法骨架的情况，应该尽量避免，但是万一前面没有考虑周全，后来出现了这种情况，怎么办呢？最好就是重构，也就是考虑修改算法骨架，尽量不要去找其他的替代方式，替代的方式也许能把功能实现了，但是会破坏整个程序的结构。

还有一种情况是既需要加入新的功能，也需要新的数据。比如，现在对于普通人员登录，要实现一个加强版，要求登录人员除了编号和密码外，还需要提供注册时留下的验证问题和验证答案，验证问题和验证答案是记录在数据库中的，不是验证码，一般 Web 开发中登录使用的验证码会放到 session 中，这里不去讨论它。

假如现在就要进行如此的扩展，应该怎样实现呢？由于需要一些其他的数据，那么就需要扩展 LoginModel，加入自己需要的数据；同时可能需要覆盖由父类提供的一些公共的方法，来实现新的功能。

还是看看代码示例吧，会比较清楚。

首先呢，需要扩展 LoginModel，把具体功能需要的数据封装起来。只是增加父类没有的数据就可以了。示例代码如下：

```
/**
 * 封装进行登录控制所需要的数据，在公共数据的基础上，
 * 添加具体模块需要的数据
 */
public class NormalLoginModel extends LoginModel{
    /**
     * 密码验证问题
     */
    private String question;
    /**
     * 密码验证答案
     */
    private String answer;
```

注意这里需要扩展公共的 LoginModel

```
    public String getQuestion() {
        return question;
    }
    public void setQuestion(String question) {
        this.question = question;
    }
    public String getAnswer() {
        return answer;
    }
    public void setAnswer(String answer) {
        this.answer = answer;
    }
}
```

其次，就是提供新的登录模块控制实现。示例代码如下：

```
/**
 * 普通用户登录控制加强版的逻辑处理
 */
public class NormalLogin2 extends LoginTemplate{
    public LoginModel findLoginUser(String loginId) {
        // 这里省略具体的处理，仅做示意，返回一个有默认数据的对象
        //注意一点：这里使用的是自己需要的数据模型了
        NormalLoginModel nlm = new NormalLoginModel();
        nlm.setLoginId(loginId);
        nlm.setPwd("testpwd");
        nlm.setQuestion("testQuestion");
        nlm.setAnswer("testAnswer");

        return nlm;
    }
    public boolean match(LoginModel lm,LoginModel dbLm){
        //这个方法需要覆盖，因为现在进行登录控制的时候，
        //需要检测4个值是否正确，而不仅仅是缺省的2个

        //先调用父类实现好的，检测编号和密码是否正确
        boolean f1 = super.match(lm, dbLm);
        if(f1){
            //如果编号和密码正确，继续检查问题和答案是否正确

            //先把数据转换成自己需要的数据
            NormalLoginModel nlm = (NormalLoginModel)lm;
```

覆盖父类方法，实现自己更多的控制

```
            NormalLoginModel dbNlm = (NormalLoginModel)dbLm;
            //检查问题和答案是否正确
            if(dbNlm.getQuestion().equals(nlm.getQuestion())
                    && dbNlm.getAnswer().equals(nlm.getAnswer())){
                return true;
            }
        }
        return false;
    }
}
```

看看这个时候的测试。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备登录人的信息
        NormalLoginModel nlm = new NormalLoginModel();
        nlm.setLoginId("testUser");
        nlm.setPwd("testpwd");
        nlm.setQuestion("testQuestion");
        nlm.setAnswer("testAnswer");
        //准备用来进行判断的对象
        LoginTemplate lt3 = new NormalLogin2();
        //进行登录测试
        boolean flag3 = lt3.login(nlm);
        System.out.println("可以进行普通人员加强版登录="+flag3);
    }
}
```

运行看看，能实现功能吗？好好测试体会一下，看看是如何扩展功能的。

16.3.2 模板的写法

在实现模板的时候，到底哪些方法实现在模板上呢？模板能不能全部实现了，也就是模板不提供抽象方法呢？当然，就算没有抽象方法，模板一样可以定义成为抽象类。

通常在模板里面包含以下操作类型。

- **模板方法**：就是定义算法骨架的方法。
- **具体的操作**：在模板中直接实现某些步骤的方法。通常这些步骤的实现算法是固定的，而且是不怎么变化的，因此可以将其当作公共功能实现在模板中。如果不需为子类提供访问这些方法的话，还可以是 private 的。这样一来，子类的实现就相对简单些。如果是子类需要访问，可以把这些方法定义为 protected final 的，因为通常情况下，这些实现不能够被子类覆盖和改变了。

- **具体的 AbstractClass 操作**：在模板中实现某些公共功能，可以提供给子类使用，一般不是具体的算法步骤的实现，而是一些辅助的公共功能。
- **原语操作**：就是在模板中定义的抽象操作，通常是模板方法需要调用的操作，是必须的操作，而且在父类中还没有办法确定下来如何实现，需要子类来真正实现的方法。
- **钩子操作**：在模板中定义，并提供默认实现的操作。这些方法通常被视为可扩展的点，但不是必须的，子类可以有选择地覆盖这些方法，以提供新的实现来扩展功能。比如，模板方法中定义了 5 步操作，但是根据需要，某种具体的实现只需要其中的 1、2、3 几个步骤，因此它就只需要覆盖实现 1、2、3 这几个步骤对应的方法。那么 4 和 5 步骤对应的方法怎么办呢，由于有默认实现，那就不用管了。也就是说钩子操作是可以被扩展的点，但不是必须的。
- **Factory Method**：在模板方法中，如果需要得到某些对象实例的话，可以考虑通过工厂方法模式来获取，把具体的构建对象的实现延迟到子类中去。

总结起来，**一个较为完整的模板定义示例**，其示例代码如下：

```
/**
 * 一个较为完整的模板定义示例
 */
public abstract class AbstractTemplate {
    /**
     * 模板方法，定义算法骨架
     */
    public final void templateMethod(){
        //第一步
        this.operation1();
        //第二步
        this.operation2();
        //第三步
        this.doPrimitiveOperation1();
        //第四步
        this.doPrimitiveOperation2();
        //第五步
        this.hookOperation1();
    }
    /**
     * 具体操作1，算法中的步骤，固定实现，而且子类不需要访问
     */
    private void operation1(){
        //在这里具体的实现
    }
    /**
```

```
     * 具体操作2，算法中的步骤，固定实现，子类可能需要访问
     * 当然也可以定义成protected的，不可以被覆盖，因此是final的
     */
    protected final void operation2(){
        //在这里具体的实现
    }
    /**
     * 具体的AbstractClass操作，子类的公共功能
     * 但通常不是具体的算法步骤
     */
    protected void commonOperation(){
        //在这里具体的实现
    }
    /**
     * 原语操作1，算法中的必要步骤，父类无法确定如何真正实现，需要子类来实现
     */
    protected abstract void doPrimitiveOperation1();
    /**
     * 原语操作2，算法中的必要步骤，父类无法确定如何真正实现，需要子类来实现
     */
    protected abstract void doPrimitiveOperation2();
    /**
     * 钩子操作，算法中的步骤，不一定需要，提供默认实现
     * 由子类选择并具体实现
     */
    protected void hookOperation1(){
        //在这里提供默认的实现
    }
    /**
     * 工厂方法，创建某个对象，这里用Object代替了，在算法实现中可能需要
     * @return 创建的某个算法实现需要的对象
     */
    protected abstract Object createOneObject();
}
```

对于上面示例的模板写法，其中定义成为 protected 的方法，可以根据需要进行调整，如果是允许所有的类都可以访问这些方法，那么可以把它们定义成为 public 的，如果只是子类需要访问这些方法，那就使用 protected 的，都是正确的写法。

16.3.3　Java 回调与模板方法模式

模板方法模式的一个目的，就在于让其他类来扩展或具体实现在模板中固定的算法骨架中的某些算法步骤。在标准的模板方法模式实现中，主要是使用继承的方式，来让父类在运行期间可以调用到子类的方法。

其实在 Java 开发中，还有另外一个方法可以实现同样的功能或是效果，那就是——Java 回调技术，通过回调在接口中定义的方法，调用到具体的实现类中的方法，其本质同样是利用 Java 的动态绑定技术。在这种实现中，可以不把实现类写成单独的类，而是使用匿名内部类来实现回调方法。

应用 Java 回调来实现模板方法模式，在实际开发中使用的也非常多，也算是模板方法模式的一种变形实现吧。

还是来示例一下，这样会更清楚。为了大家更好地对比理解，把前面用标准模板方法模式实现的例子，采用 Java 回调来实现一下。

（1）先定义一个模板方法需要的回调接口。

在这个接口中需要把所有可以被扩展的方法都要定义出来。实现的时候，可以不扩展，直接转调模板中的默认实现，但是不能不定义出来，因为是接口，不定义出来，对于想要扩展这些功能的地方就没有办法了。示例代码如下：

```
/**
 * 登录控制的模板方法需要的回调接口，需要把所有需要的接口方法都定义出来
 * 或者说是所有可以被扩展的方法都需要被定义出来
 */
public interface LoginCallback {
    /**
     * 根据登录编号来查找和获取存储中相应的数据
     * @param loginId 登录编号
     * @return 登录编号在存储中相对应的数据
     */
    public LoginModel findLoginUser(String loginId);
    /**
     * 对密码数据进行加密
     * @param pwd 密码数据
     * @param template LoginTemplate对象，通过它来调用在
     *              LoginTemplate中定义的公共方法或默认实现
     * @return 加密后的密码数据
     */
    public String encryptPwd(String pwd,LoginTemplate template);
    /**
     * 判断用户填写的登录数据和存储中对应的数据是否匹配的上
     * @param lm 用户填写的登录数据
```

```
     * @param dbLm 在存储中对应的数据
     * @param template LoginTemplate对象，通过它来调用在
     * LoginTemplate中定义的公共方法或默认实现
     * @return true表示匹配成功，false表示匹配失败
     */
    public boolean match(LoginModel lm,LoginModel dbLm
                                  ,LoginTemplate template);
}
```

（2）这里使用的 LoginModel 跟以前相比没有任何变化，就不再赘述。

（3）下面来定义登录控制的模板。它的变化相对较多，大致有以下一些。

- 不再是抽象的类了，所有的抽象方法都删除了。
- 对模板方法，就是 login 的那个方法，添加一个参数，传入回调接口。
- 在模板方法实现中，除了在模板中固定的实现外，所有可以被扩展的方法，都应该通过回调接口进行调用。

示例代码如下：

```
/**
 *  登录控制的模板
 */
public class LoginTemplate {
    /**
     * 判断登录数据是否正确，也就是是否能登录成功
     * @param lm 封装登录数据的Model
     * @param callback LoginCallback对象
     * @return true表示登录成功，false表示登录失败
     */
    public final boolean login(LoginModel lm,LoginCallback callback){
        //1：根据登录人员的编号去获取相应的数据
        LoginModel dbLm = callback.findLoginUser(lm.getLoginId());
        if(dbLm!=null){
            //2：对密码进行加密
            String encryptPwd =
                        callback.encryptPwd(lm.getPwd(),this);
            //把加密后的密码设置回到登录数据模型中
            lm.setPwd(encryptPwd);
            //3：判断是否匹配
            return callback.match(lm, dbLm,this);
        }
        return false;
    }
```

```
    /**
     * 对密码数据进行加密
     * @param pwd 密码数据
     * @return 加密后的密码数据
     */
    public String encryptPwd(String pwd){
        return pwd;
    }
    /**
     * 判断用户填写的登录数据和存储中对应的数据是否匹配得上
     * @param lm 用户填写的登录数据
     * @param dbLm 在存储中对应的数据
     * @return true表示匹配成功，false表示匹配失败
     */
    public boolean match(LoginModel lm,LoginModel dbLm){
        if(lm.getLoginId().equals(dbLm.getLoginId())
                && lm.getPwd().equals(dbLm.getPwd())){
            return true;
        }
        return false;
    }
}
```

（4）由于是直接在调用的地方传入回调的实现，通常可以通过匿名内部类的方式来实现回调接口，当然实现成为具体类也是可以的。如果采用匿名内部类的方式来使用模板，那么就不需要原来的 NormalLogin 和 WorkerLogin 了。

（5）写个客户端来测试看看。客户端需要使用匿名内部类来实现回调接口，并实现其中想要扩展的方法。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备登录人的信息
        LoginModel lm = new LoginModel();
        lm.setLoginId("admin");
        lm.setPwd("workerpwd");
        //准备用来进行判断的对象
        LoginTemplate lt = new LoginTemplate();

        //进行登录测试，先测试普通人员登录
        boolean flag = lt.login(lm,new LoginCallback(){
            public String encryptPwd(String pwd
```

```
                                , LoginTemplate template) {
            //自己不需要实现这个功能，直接转调模板中的默认实现
            return template.encryptPwd(pwd);
        }
        public LoginModel findLoginUser(String loginId) {
            // 这里省略具体的处理，仅做示意，返回一个有默认数据的对象
            LoginModel lm = new LoginModel();
            lm.setLoginId(loginId);
            lm.setPwd("testpwd");
            return lm;
        }
        public boolean match(LoginModel lm, LoginModel dbLm,
                LoginTemplate template) {
            //自己不需要覆盖，直接转调模板中的默认实现
            return template.match(lm, dbLm);
        }
    });
    System.out.println("可以进行普通人员登录="+flag);

    //测试工作人员登录
    boolean flag2 = lt.login(lm,new LoginCallback(){
        public String encryptPwd(String pwd
                                , LoginTemplate template) {
            //覆盖父类的方法，提供真正的加密实现
            //这里对密码进行加密，比如使用MD5、3DES等，省略了
            System.out.println("使用MD5进行密码加密");
            return pwd;
        }
        public LoginModel findLoginUser(String loginId) {
            // 这里省略具体的处理，仅做示意，返回一个有默认数据的对象
            LoginModel lm = new LoginModel();
            lm.setLoginId(loginId);
            lm.setPwd("workerpwd");
            return lm;
        }
        public boolean match(LoginModel lm, LoginModel dbLm,
                LoginTemplate template) {
            //自己不需要覆盖，直接转调模板中的默认实现
            return template.match(lm, dbLm);
        }
```

```
        });
        System.out.println("可以登录工作平台="+flag2);
    }
}
```

运行一下，看看效果是不是跟前面采用继承的方式实现的结果是一样的，然后好好比较一下这两种实现方式。

（6）简单小结一下对于模板方法模式的这两种实现方式：

- 使用继承的方式，抽象方法和具体实现的关系是在编译期间静态决定的，是类级的关系；使用 Java 回调，这个关系是在运行期间动态决定的，是对象级的关系。
- 相对而言，使用回调机制会更灵活，因为 Java 是单继承的，如果使用继承的方式，对于子类而言，今后就不能继承其他对象了，而使用回调，是基于接口的。
- 相对而言，使用继承方式会更简单点，因为父类提供了实现的方法，子类如果不想扩展，那就不用管。如果使用回调机制，回调的接口需要把所有可能被扩展的方法都定义进去，这就导致实现的时候，不管你要不要扩展，都要实现这个方法，哪怕你什么都不做，只是转调模板中已有的实现，都要写出来。

延伸

从另一方面说，回调机制是通过委托的方式来组合功能，它的耦合强度要比继承低一些，这会给我们更多的灵活性。比如某些模板实现的方法，在回调实现的时候可以不调用模板中的方法，而是调用其他实现中的某些功能，也就是说功能不再局限在模板和回调实现上了，可以更灵活地组织功能。

事实上，在前面讲命令模式的时候也提到了 Java 回调，还通过退化命令模式来实现了 Java 回调的功能。所以也有这样的说法：命令模式可以作为模板方法模式的一种替代实现，那就是因为可以使用 Java 回调来实现模板方法模式。

16.3.4　典型应用：排序

模板方法模式的一个非常典型的应用，就是实现排序的功能。至于有些朋友认为排序是策略模式的体现，这很值得商榷。先来看看在 Java 中排序功能的实现，然后再来说明为什么排序的实现主要体现了模板方法模式，而非策略模式。

在 java.util 包中，有一个 Collections 类，它里面实现了对列表排序的功能，提供了一个静态的 sort 方法，接受一个列表和一个 Comparator 接口的实例，这个方法实现的大致步骤如下。

（1）先把列表转换成为对象数组。

（2）通过 Arrays 的 sort 方法来对数组进行排序，传入 Comparator 接口的实例。

（3）然后再把排好序的数组的数据设置回到原来的列表对象中去 。

这其中的算法步骤是固定的，也就是算法骨架是固定的了，只是其中具体比较数据大小的步骤，需要由外部来提供，也即是传入的 Comparator 接口的实例，就是用来实现数据比较的，在算法内部会通过这个接口来回调具体的实现。

如果 Comparator 接口的 compare()方法返回一个小于 0 的数，表示被比较的两个对象中，前面的对象小于后面的对象；如果返回一个等于 0 的数，表示被比较的两个对象相等；如果返回一个大于 0 的数，表示被比较的两个对象中，前面的对象大于后面的对象。

下面一起看看使用 Collections 来对列表进行排序的例子。假如现在要实现对一个拥有多个用户数据模型的列表进行排序。

（1）先来定义出封装用户数据的对象模型。示例代码如下：

```
/**
 * 用户数据模型
 */
public class UserModel {
    private String userId,name;
    private int age;
    public UserModel(String userId,String name,int age) {
        this.userId = userId;
        this.name = name;
        this.age = age;
    }
    public String getUserId() {
        return userId;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public String toString(){
        return "userId="+userId+",name="+name+",age="+age;
    }
}
```

（2）直接使用 Collections 来排序。写个客户端来测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备要测试的数据
        UserModel um1 = new UserModel("u1","user1",23);
        UserModel um2 = new UserModel("u2","user2",22);
        UserModel um3 = new UserModel("u3","user3",21);
        UserModel um4 = new UserModel("u4","user4",24);
        //添加到列表中
```

```
        List<UserModel> list = new ArrayList<UserModel>();
        list.add(um1);
        list.add(um2);
        list.add(um3);
        list.add(um4);

        System.out.println("排序前---------------------〉 ");
        printList(list);
        //实现比较器，也可以单独用一个类来实现
        Comparator c = new Comparator(){
            public int compare(Object obj1, Object obj2) {
                //假如实现按照年龄升序排序
                UserModel tempUm1 = (UserModel)obj1;
                UserModel tempUm2 = (UserModel)obj2;
                if(tempUm1.getAge() > tempUm2.getAge()){
                    return 1;
                }else if(tempUm1.getAge() == tempUm2.getAge()){
                    return 0;
                }else if(tempUm1.getAge() < tempUm2.getAge()){
                    return -1;
                }
                return 0;
            }};

        //排序
        Collections.sort(list,c);

        System.out.println("排序后---------------------〉 ");
        printList(list);
    }
    private static void printList(List<UserModel> list){
        for(UserModel um : list){
            System.out.println(um);
        }
    }
}
```

运行结果如下所示：

```
排序前---------------------〉
userId=u1,name=user1,age=23
userId=u2,name=user2,age=22
```

```
userId=u3,name=user3,age=21
userId=u4,name=user4,age=24
排序后--------------------〉
userId=u3,name=user3,age=21
userId=u2,name=user2,age=22
userId=u1,name=user1,age=23
userId=u4,name=user4,age=24
```

（3）小结。

看了上面的示例，你会发现，究竟列表会按照什么标准来排序，完全是依靠 Comparator 的具体实现，上面实现的是按照年龄的升序排列，你也可以尝试修改这个排序的比较器，那么得到的结果就会不一样了。

也就是说，排序的算法是已经固定了的，只是进行排序比较的这一个步骤，由外部来实现。我们可以通过修改这个步骤的实现，从而实现不同的排序方式。因此从排序比较这个功能来看，是策略模式的体现。

> 注意 **但是请注意一点，你只是修改了排序的比较方式，并不是修改了整个排序的算法，事实上，现在 Collections 的 sort()方法使用的是合并排序的算法，无论你怎样修改比较器的实现，sort()方法实现的算法是不会改变的，不可能变成冒泡排序或是其他的排序算法。**

（4）排序，到底是模板方法模式的实例，还是策略模式的实例，到底哪个说法更合适？

认为是策略模式的实例的理由：

- 上面的排序实现，并没有像标准的模板方法模式那样，使用子类来扩展父类，至少从表面上看不太像模板方法模式；
- 排序使用的 Comparator 的实例，可以看成是不同的算法实现，在具体排序时，会选择使用不同的 Comparator 实现，就相当于是在切换算法的实现。

因此认为排序是策略模式的实例。

认为是模板方法模式的实例的理由：

- 首模板方法模式的本质是固定算法骨架，虽然使用继承是标准的实现方式，但是通过回调来实现，也不能说这就不是模板方法模式；
- 从整体程序上看，排序的算法并没有改变，不过是某些步骤的实现发生了变化，也就是说通过 Comparator 来切换的是不同的比较大小的实现，相对于整个排序算法而言，它不过是其中的一个步骤而已。

因此认为是模板方法模式的实例。

总结语：

排序的实现，实际上组合使用了模板方法模式和策略模式，从整体来看是模板方法模式，但到了局部，比如排序比较算法的实现上，就使用的是策略模式了。

至于排序具体属于谁的实例，这或许是个仁者见仁、智者见智的事情，我们倾向于

说：排序是模板方法模式的实例。**毕竟设计模式的东西，要从整体上、设计上、本质上去看待问题，而不能从表面上或者是局部来看待问题。**

16.3.5　实现通用的增删改查

对于实现通用的增删改查的功能，基本上是每个做企业级应用系统的公司都有的功能，实现的方式也是多种多样的，一种很常见的设计就是泛型加上模板方法模式，再加上使用 Java 回调技术，尤其是在使用 Spring 和 Hibernate 等流行框架的应用系统中更是常见。

> **注意** 为了突出主题，以免分散大家的注意力，我们不去使用 Spring 和 Hibernate 这样的流行框架，也不去使用泛型，只用模板方法模式来实现一个简单的、用 JDBC 实现的通用增删改查的功能。

先在数据库中定义一个演示用的表，演示用的是 Oracle 数据库。其实你可以用任意的数据库，只是数据类型要做相应的调整。简单的数据字典如下：表名是 tbl_user。

字 段	名 称	类型、长度	主外键
Uuid	编号	Varchar2(10)	主键
Name	姓名	Varchar2(20)	
Age	年龄	Number(3)	

（1）定义相应的数据对象来描述数据。示例代码如下：

```
/**
 * 描述用户的数据模型
 */
public class UserModel {
    private String uuid;
    private String name;
    private int age;
    public String getUuid() {
        return uuid;
    }
    public void setUuid(String uuid) {
        this.uuid = uuid;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
```

就是简单地定义出描述数据的属性，并提供相应的 getter/setter 方法

```
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String toString(){
        return "uuid="+uuid+",name="+name+",age="+age;
    }
}
```

（2）定义一个用于封装通用查询数据的查询用的数据模型。由于这个查询数据模型和上面定义的数据模型有很大一部分是相同的，因此让这个查询模型继承上面的数据模型，然后添加上多出来的查询条件。示例代码如下：

```
/**
 * 描述查询用户的条件数据的模型
 */




    public int getAge2() {
        return age2;
    }
    public void setAge2(int age2) {
        this.age2 = age2;
    }
}
```

（3）为了让大家能更好的理解这个通用的实现，先不去使用模板方法模式，直接使用 JDBC 来实现增删改查的功能。

所有的方法都需要和数据库进行连接，因此先把和数据库连接的公共方法定义出来。没有使用连接池，用最简单的 JDBC 自己连接，示例代码如下：

```
    /**
     * 获取与数据库的连接
     * @return 数据库连接
```

```
     * @throws Exception
     */
    private Connection getConnection()throws Exception{
        Class.forName("你用的数据库对应的JDBC驱动类");
        return DriverManager.getConnection(
                "连接数据库的URL",
                "用户名","密码");
    }
```

使用纯 JDBC 来实现新增的功能。示例代码如下：

```
    public void create(UserModel um) {
        Connection conn = null;
        try {
            conn = this.getConnection();
            String sql = "insert into tbl_user values(?,?,?)";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, um.getUuid());
            pstmt.setString(2, um.getName());
            pstmt.setInt(3, um.getAge());

            pstmt.executeUpdate();

            pstmt.close();
        } catch (Exception err) {
            err.printStackTrace();
        } finally {
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

修改和删除的功能和新增功能差不多，只是 sql 不同，还有设置 sql 中变量值不同，这里就不去写了。

接下来看看查询方面的功能。查询方面只做一个通用的查询实现，其他查询的实现基本上也差不多。示例代码如下：

```
    public Collection getByCondition(UserQueryModel uqm){
        Collection col = new ArrayList();
        Connection conn = null;
```

```
        try{
            conn = this.getConnection();
            String sql = "select * from tbl_user where 1=1 ";
            sql = this.prepareSql(sql, uqm);
            PreparedStatement pstmt = conn.prepareStatement(sql);
            this.setValue(pstmt, uqm);
            ResultSet rs = pstmt.executeQuery();
            while(rs.next()){
                col.add(this.rs2Object(rs));
            }
           rs.close();
           pstmt.close();
        }catch(Exception err){
            err.printStackTrace();
        }finally{
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return col;
    }
    /**
     * 为通用查询动态地拼接sql的条件部分，基本思路是：
     * 如果用户填写了相应的条件，那么才在sql中添加对应的条件
     * @param sql sql的主干部分
     * @param uqm 封装查询条件的数据模型
     * @return 拼接好的sql语句
     */
    private String prepareSql(String sql,UserQueryModel uqm){
        StringBuffer buffer = new StringBuffer();
        buffer.append(sql);
        //绝对匹配
        if(uqm.getUuid()!=null&& uqm.getUuid().trim().length()>0){
            buffer.append(" and uuid=? ");
        }
```

```
        //模糊匹配
        if(uqm.getName()!=null&& uqm.getName().trim().length()>0){
            buffer.append(" and name like ? ");
        }
        //区间匹配
        if(uqm.getAge() > 0){
            buffer.append(" and age >=? ");
        }
        if(uqm.getAge2() > 0){                          区间匹配
            buffer.append(" and age <=? ");
        }
        return buffer.toString();
    }
    /**
     * 为通用查询的sql动态设置条件的值
     * @param pstmt 预处理查询sql的对象
     * @param uqm 封装查询条件的数据模型
     * @throws Exception
     */
    private void setValue(PreparedStatement pstmt
                         ,UserQueryModel uqm)throws Exception{
        int count = 1;
        if(uqm.getUuid()!=null
                && uqm.getUuid().trim().length()>0){
            pstmt.setString(count, uqm.getUuid());
            count++;
        }
        if(uqm.getName()!=null
                && uqm.getName().trim().length()>0){
            pstmt.setString(count, "%"+uqm.getName()+"%");
            count++;
        }
        if(uqm.getAge() > 0){
            pstmt.setInt(count, uqm.getAge());
            count++;
        }
        if(uqm.getAge2() > 0){
            pstmt.setInt(count, uqm.getAge2());
            count++;
        }
```

```
    }
    /**
     * 把查询返回的结果集转换成为对象
     * @param rs 查询返回的结果集
     * @return 查询返回的结果集转换成为对象
     * @throws Exception
     */
    private UserModel rs2Object(ResultSet rs)throws Exception{
        UserModel um = new UserModel();
        String uuid = rs.getString("uuid");
        String name = rs.getString("name");
        int age = rs.getInt("age");

        um.setAge(age);
        um.setName(name);
        um.setUuid(uuid);

        return um;
    }
```

（4）基本的 JDBC 实现写完了，该来看看如何把模板方法模式用上了。模板方法是要定义算法的骨架，而具体步骤的实现还是由子类来完成，因此把固定的算法骨架抽取出来，就成了使用模板方法模式的重点了。

首先来观察新增、修改、删除的功能，发现哪些是固定的，哪些是变化的呢？分析发现变化的只有 sql 语句，还有为 sql 中的“？”设置值的语句，真正执行 sql 的过程是差不多的，是不变化的。

再来观察查询的方法，查询的过程是固定的。变化的除了有 sql 语句、为 sql 中的“？”设置值的语句之外，还多了一个如何把查询回来的结果集转换成对象集的实现。

好了，找到变与不变之处，就可以来设计模板了。先定义出增删改查各自的实现步骤来，也就是定义好各自的算法骨架，然后把变化的部分定义成为原语操作或钩子操作，如果一定要子类实现的那就定义成为原语操作；在模板中提供默认实现，且不强制子类实现的功能定义成为钩子操作就可以了。

另外，来回需要传递数据，由于是通用的方法，就不能用具体的类型了，又不考虑泛型，那么就定义成 Object 类型好了。

根据上面的思路，一个简单的、能实现对数据进行增删改查的模板就可以实现出来了。完整的示例代码如下：

```
/**
```

```
 * 一个简单的实现JDBC增删改查功能的模板
 */
public abstract class JDBCTemplate {
    /**
     * 定义当前的操作类型是新增
     */
    protected final static int TYPE_CREATE = 1;
    /**
     * 定义当前的操作类型是修改
     */
    protected final static int TYPE_UPDATE = 2;
    /**
     * 定义当前的操作类型是删除
     */
    protected final static int TYPE_DELETE = 3;
    /**
     * 定义当前的操作类型是按条件查询
     */
    protected final static int TYPE_CONDITION = 4;

/*---------------------模板方法---------------------*/
    /**
     * 实现新增的功能
     * @param obj 需要被新增的数据对象
     */
    public final void create(Object obj){
        //1：获取新增的sql
        String sql = this.getMainSql(TYPE_CREATE);
        //2：调用通用的更新实现
        this.executeUpdate(sql, TYPE_CREATE,obj);
    }
    /**
     * 实现修改的功能
     * @param obj 需要被修改的数据对象
     */
    public final void update(Object obj){
        //1：获取修改的sql
        String sql = this.getMainSql(TYPE_UPDATE);
        //2：调用通用的更新实现
        this.executeUpdate(sql, TYPE_UPDATE,obj);
```

```
    }
    /**
     * 实现删除的功能
     * @param obj 需要被删除的数据对象
     */
    public final void delete(Object obj){
        //1: 获取删除的sql
        String sql = this.getMainSql(TYPE_DELETE);
        //2: 调用通用的更新实现
        this.executeUpdate(sql, TYPE_DELETE,obj);
    }
    /**
     * 实现按照条件查询的功能
     * @param qm 封装查询条件的数据对象
     * @return 符合条件的数据对象集合
     */
    public final Collection getByCondition(Object qm){
        //1: 获取查询的sql
        String sql = this.getMainSql(TYPE_CONDITION);
        //2: 调用通用的查询实现
        return this.getByCondition(sql, qm);
    }

/*---------------------原语操作---------------------*/
    /**
     * 获取操作需要的主干sql
     * @param type 操作类型
     * @return 操作对应的主干sql
     */
    protected abstract String getMainSql(int type);
    /**
     * 为更新操作的sql中的“?”设置值
     * @param type 操作类型
     * @param pstmt PreparedStatement对象
     * @param obj 操作的数据对象
     * @throws Exception
     */
    protected abstract void setUpdateSqlValue(int type,
            PreparedStatement pstmt,Object obj) throws Exception;
    /**
```

```
     * 为通用查询动态地拼接sql的条件部分，基本思路是：
     * 只有用户填写了相应的条件，那么才在sql中添加对应的条件
     * @param sql sql的主干部分
     * @param qm 封装查询条件的数据模型
     * @return 拼接好的sql语句
     */
    protected abstract String prepareQuerySql(String sql,Object qm);
    /**
     * 为通用查询的sql动态设置条件的值
     * @param pstmt 预处理查询sql的对象
     * @param qm 封装查询条件的数据模型
     * @throws Exception
     */
    protected abstract void setQuerySqlValue(
            PreparedStatement pstmt,Object qm)throws Exception;
    /**
     * 把查询返回的结果集转换成为的数据对象
     * @param rs 查询返回的结果集
     * @return 查询返回的结果集转换成为数据对象
     * @throws Exception
     */
    protected abstract Object rs2Object(ResultSet rs)throws
Exception;

/*---------------------钩子操作---------------------*/
    /**
     * 连接数据库的默认实现，可以被子类覆盖
     * @return 数据库连接
     * @throws Exception
     */
    protected Connection getConnection()throws Exception{
        Class.forName("你用的数据库对应的JDBC驱动类");
        return DriverManager.getConnection(
                "连接数据库的URL",
                "用户名","密码");
    }
    /**
     * 执行查询
     * @param sql 查询的主干sql语句
     * @param qm 封装查询条件的数据模型
```

```
     * @return 查询后的结果对象集合
     */
    protected Collection getByCondition(String sql,Object qm){
        Collection col = new ArrayList();
        Connection conn = null;
        try{
            //调用钩子方法
            conn = this.getConnection();
            //调用原语操作
            sql = this.prepareQuerySql(sql, qm);
            PreparedStatement pstmt = conn.prepareStatement(sql);
            //调用原语操作
            this.setQuerySqlValue(pstmt, qm);
            ResultSet rs = pstmt.executeQuery();
            while(rs.next()){
                //调用原语操作
                col.add(this.rs2Object(rs));
            }

            rs.close();
            pstmt.close();
        }catch(Exception err){
            err.printStackTrace();
        }finally{
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return col;
    }
    /**
     * 执行更改数据的sql语句，包括增删改的功能
     * @param sql 需要执行的sql语句
     * @param callback 回调接口，回调为sql语句赋值的方法
     */
    protected void executeUpdate(String sql,int type,Object obj){
        Connection conn = null;
        try{
```

```
            //调用钩子方法
            conn = this.getConnection();
            PreparedStatement pstmt = conn.prepareStatement(sql);
            //调用原语操作
            this.setUpdateSqlValue(type,pstmt,obj);
            pstmt.executeUpdate();
            pstmt.close();
        }catch(Exception err){
            err.printStackTrace();
        }finally{
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

（5）简单又可以通用的 JDBC 模板做好了，下面看看如何使用这个模板来实现具体的增删改查功能。示例代码如下：

```
/**
 * 具体的实现用户管理的增删改查功能
 */
public class UserJDBC extends JDBCTemplate{
    protected String getMainSql(int type) {
        //根据操作类型，返回相应的主干sql语句
        String sql = "";
        if(type == TYPE_CREATE){
            sql = "insert into tbl_user values(?,?,?)";
        }else if(type == TYPE_DELETE){
            sql = "delete from tbl_user where uuid=?";
        }else if(type == TYPE_UPDATE){
            sql = "update tbl_user set name=?,age=? where uuid=?";
        }else if(type == TYPE_CONDITION){
            sql = "select * from tbl_user where 1=1 ";
        }
        return sql;
    }
    protected void setUpdateSqlValue(int type
        , PreparedStatement pstmt,Object obj) throws Exception{
```

```
        //设置增、删、改操作的sql中“?”对应的值
        if(type == TYPE_CREATE){
            this.setCreateValue(pstmt, (UserModel)obj);
        }else if(type == TYPE_DELETE){
            this.setDeleteValue(pstmt, (UserModel)obj);
        }else if(type == TYPE_UPDATE){
            this.setUpdateValue(pstmt, (UserModel)obj);
        }
    }
    protected Object rs2Object(ResultSet rs)throws Exception{
        UserModel um = new UserModel();
        String uuid = rs.getString("uuid");
        String name = rs.getString("name");
        int age = rs.getInt("age");

        um.setAge(age);
        um.setName(name);
        um.setUuid(uuid);

        return um;
    }
    protected String prepareQuerySql(String sql,Object qm){
        UserQueryModel uqm = (UserQueryModel)qm;
        StringBuffer buffer = new StringBuffer();
        buffer.append(sql);
        if(uqm.getUuid()!=null&& uqm.getUuid().trim().length()>0){
            buffer.append(" and uuid=? ");
        }
        if(uqm.getName()!=null&& uqm.getName().trim().length()>0){
            buffer.append(" and name like ? ");
        }
        if(uqm.getAge() > 0){
            buffer.append(" and age >=? ");
        }
        if(uqm.getAge2() > 0){
            buffer.append(" and age <=? ");
        }
        return buffer.toString();
    }
    protected void setQuerySqlValue(PreparedStatement pstmt
```

这个方法跟以前的实现一样

```
                                    ,Object qm)throws Exception{
        UserQueryModel uqm = (UserQueryModel)qm;
        int count = 1;
        if(uqm.getUuid()!=null&& uqm.getUuid().trim().length()>0){
            pstmt.setString(count, uqm.getUuid());
            count++;
        }
        if(uqm.getName()!=null&& uqm.getName().trim().length()>0){
            pstmt.setString(count, "%"+uqm.getName()+"%");
            count++;
        }
        if(uqm.getAge() > 0){
            pstmt.setInt(count, uqm.getAge());
            count++;
        }
        if(uqm.getAge2() > 0){
            pstmt.setInt(count, uqm.getAge2());
            count++;
        }
    }
    private void setCreateValue(PreparedStatement pstmt,UserModel
                                        um)throws Exception{
        pstmt.setString(1, um.getUuid());
        pstmt.setString(2, um.getName());
        pstmt.setInt(3, um.getAge());
    }
    private void setUpdateValue(PreparedStatement pstmt,UserModel
                                        um)throws Exception{
        pstmt.setString(1, um.getName());
        pstmt.setInt(2, um.getAge());
        pstmt.setString(3, um.getUuid());
    }
    private void setDeleteValue(PreparedStatement pstmt,UserModel
                                        um)throws Exception{
        pstmt.setString(1, um.getUuid());
    }
}
```

看到这里，可能有些朋友会想，为何不把准备 sql 的方法为 sql 中“？”赋值的方法，还有结果集映射成为对象的方法也做成公共的呢？

注意	**其实这些方法是可以考虑做成公共的，用反射机制就可以实现，但是这里为了突出模板方法模式的使用，以免加的东西太多，把大家搞迷惑了。** **事实上，用模板方法加上泛型再加上反射的技术，就可以实现可重用的，使用模板时几乎不用再写代码的数据层实现，这里就不去展开了。**

（6）享受的时刻到了，来写个客户端，使用 UserJDBC 的实现。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        UserJDBC uj = new UserJDBC();
        //先新增几条
        UserModel um1 = new UserModel();
        um1.setUuid("u1");
        um1.setName("张三");
        um1.setAge(22);
        uj.create(um1);

        UserModel um2 = new UserModel();
        um2.setUuid("u2");
        um2.setName("李四");
        um2.setAge(25);
        uj.create(um2);

        UserModel um3 = new UserModel();
        um3.setUuid("u3");
        um3.setName("王五");
        um3.setAge(32);
        uj.create(um3);

        //测试修改
        um3.setName("王五被改了");
        um3.setAge(35);
        uj.update(um3);

        //测试查询
        UserQueryModel uqm = new UserQueryModel();
        uqm.setAge(20);
        uqm.setAge2(36);
        Collection<UserModel> col = uj.getByCondition(uqm);
        for(UserModel tempUm : col){
```

```
            System.out.println(tempUm);
        }
    }
}
```

运行一下，看看结果，看看数据库的值，再好好体会一下是如何实现的。

16.3.6　模板方法模式的优缺点

- 模板方法模式的优点是实现代码复用。

  模板方法模式是一种实现代码复用的很好的手段。通过把子类的公共功能提炼和抽取，把公共部分放到模板中去实现。

- 模板方法模式的缺点是算法骨架不容易升级。

  模板方法模式最基本的功能就是通过模板的制定，把算法骨架完全固定下来。事实上模板和子类是非常耦合的，如果要对模板中的算法骨架进行变更，可能就会要求所有相关的子类进行相应的变化。所以抽取算法骨架的时候要特别小心，尽量确保是不会变化的部分才放到模板中。

16.3.7　思考模板方法模式

1. 模板方法模式的本质

> 模板方法模式的本质：**固定算法骨架。**

模板方法模式主要是通过制定模板，把算法步骤固定下来，至于谁来实现，模板可以自己提供实现，也可以由子类去实现，还可以通过回调机制让其他类来实现。

通过固定算法骨架来约束子类的行为，并在特定的扩展点来让子类进行功能扩展，从而让程序既有很好的复用性，又有较好的扩展性。

2. 对设计原则的体现

模板方法很好地体现了开闭原则和里氏替换原则。

首先从设计上分离变与不变，然后把不变的部分抽取出来，定义到父类中，比如算法骨架，一些公共的、固定的实现等。这些不变的部分被封闭起来，尽量不去修改它们。要想扩展新的功能，那就使用子类来扩展，通过子类来实现可变化的步骤，对于这种新增功能的做法是开放的。

其次，能够实现统一的算法骨架，通过切换不同的具体实现来切换不同的功能，一个根本原因就是里氏替换原则，遵循这个原则，保证所有的子类实现的是同一个算法模板，并能在使用模板的地方，根据需要切换不同的具体实现。

3. 何时选用模板方法模式

建议在以下情况中选用模板方法模式。

- 需要固定定义算法骨架，实现一个算法的不变的部分，并把可变的行为留给子类来实现的情况。
- 各个子类中具有公共行为，应该抽取出来，集中在一个公共类中去实现，从而避免代码重复。
- 需要控制子类扩展的情况。模板方法模式会在特定的点来调用子类的方法，这样只允许在这些点进行扩展。

16.3.8 相关模式

- 模板方法模式和工厂方法模式

  这两个模式可以配合使用。

  模板方法模式可以通过工厂方法来获取需要调用的对象。

- 模板方法模式和策略模式

  这两个模式的功能有些相似，但是是有区别的。

  从表面上看，两个模式都能实现算法的封装，但是模板方法封装的是算法的骨架，这个算法骨架是不变的，变化的是算法中某些步骤的具体实现；而策略模式是把某个步骤的具体实现算法封装起来，所有封装的算法对象是等价的，可以相互替换。

  因此，可以在模板方法中使用策略模式，就是把那些变化的算法步骤通过使用策略模式来实现，但是具体选取哪个策略还是要由外部来确定，而整体的算法步骤，也就是算法骨架则由模板方法来定义了。

第 17 章　策略模式（Strategy）

17.1 场景问题

17.1.1 报价管理

向客户报价，对于销售部门的人来讲，这是一个非常重大和复杂的问题，对不同的客户要报不同的价格，比如：

- 对普通客户或者是新客户报的是全价；
- 对老客户报的价格，根据客户年限，给予一定的折扣；
- 对大客户报的价格，根据大客户的累计消费金额，给予一定的折扣；
- 还要考虑客户购买的数量和金额，比如，虽然是新用户，但是一次购买的数量非常大，或者是总金额非常高，也会有一定的折扣；
- 还有，报价人员的职务高低，也决定了他是否有权限对价格进行一定的浮动折扣。

甚至在不同的阶段，对客户的报价也不同。一般情况是刚开始比较高，越接近成交阶段，报价越趋于合理。

总之，向客户报价是非常复杂的，因此在一些 CRM（客户关系管理）系统中，会有一个单独的报价管理模块，来处理复杂的报价功能。

为了演示的简洁性，假定现在需要实现一个简化的报价管理，实现如下的功能。

- 对普通客户或者是新客户报全价；
- 对老客户报的价格，统一折扣 5%；
- 对大客户报的价格，统一折扣 10%。

该怎么实现呢？

17.1.2 不用模式的解决方案

要实现对不同的人员报不同价格的功能，无外乎就是判断起来麻烦点，也没多难，很快就有朋友能写出如下的实现代码。示例代码如下：

```
/**
 * 价格管理，主要完成计算向客户所报价格的功能
 */
public class Price {
    /**
     * 报价，对不同类型的，计算不同的价格
     * @param goodsPrice 商品销售原价
     * @param customerType 客户类型
     * @return 计算出来的，应该给客户报的价格
     */
    public double quote(double goodsPrice,String customerType){
        if("普通客户".equals(customerType)){
```

```
            System.out.println("对于新客户或者是普通客户，没有折扣");
            return goodsPrice;
        }else if("老客户".equals(customerType)){
            System.out.println("对于老客户，统一折扣5%");
            return goodsPrice*(1-0.05);
        }else if("大客户".equals(customerType)){
            System.out.println("对于大客户，统一折扣10%");
            return goodsPrice*(1-0.1);
        }
        //其余人员都是报原价
        return goodsPrice;
    }
}
```

17.1.3　有何问题

上面的写法是很简单的，也很容易想到，但是仔细想想，这样实现，问题可不小，有以下两个问题。

（1）价格类包含了所有计算报价的算法，使得价格类，尤其是报价这个方法比较庞杂，难以维护。

有朋友可能会想，这很简单嘛，把这些算法从报价方法里面拿出去，形成独立的方法不就可以解决这个问题了吗？据此写出如下的实现代码。示例代码如下：

```
/**
 * 价格管理，主要完成计算向客户所报价格的功能
 */
public class Price {
    /**
     * 报价，对不同类型的，计算不同的价格
     * @param goodsPrice 商品销售原价
     * @param customerType 客户类型
     * @return 计算出来的，应该给客户报的价格
     */
    public double quote(double goodsPrice,String customerType){
        if("普通客户".equals(customerType)){
            return this.calcPriceForNormal(goodsPrice);
        }else if("老客户".equals(customerType)){
            return this.calcPriceForOld(goodsPrice);
        }else if("大客户".equals(customerType)){
            return this.calcPriceForLarge(goodsPrice);
```

```
        }
        //其余人员都是报原价
        return goodsPrice;
    }
    /**
     * 为新客户或者是普通客户计算应报的价格
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    private double calcPriceForNormal(double goodsPrice){
        System.out.println("对于新客户或者是普通客户，没有折扣");
        return goodsPrice;
    }
    /**
     * 为老客户计算应报的价格
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    private double calcPriceForOld(double goodsPrice){
        System.out.println("对于老客户，统一折扣5%");
        return goodsPrice*(1-0.05);
    }
    /**
     * 为大客户计算应报的价格
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    private double calcPriceForLarge(double goodsPrice){
        System.out.println("对于大客户，统一折扣10%");
        return goodsPrice*(1-0.1);
    }
}
```

这样看起来，比刚开始稍稍好点，计算报价的方法也稍稍简单一点，这样维护起来也稍好一些，某个算法发生了变化，直接修改相应的私有方法就可以了。扩展起来也容易一点，比如要增加一个“战略合作客户”的类型，报价为直接 8 折，就只需要在价格类里面新增加一个私有的方法来计算新的价格，然后在计算报价的方法中新添一个 else-if 即可。看起来似乎很不错了。

真的很不错了吗？

再想想，问题还是存在，只不过从计算报价的方法挪动到价格类中了，假如有 100

个或者更多这样的计算方式，会使这个价格类非常庞大，难以维护。而且，维护和扩展都需要去修改已有的代码，这是很不好的，违反了开—闭原则。

（2）经常会有这样的需要，在不同的时候，要使用不同的计算方式。

比如，在公司周年庆的时候，所有的客户额外增加 3%的折扣；在换季促销的时候，普通客户是额外增加折扣 2%，老客户是额外增加折扣 3%，大客户是额外增加折扣 5%。这意味着计算报价的方式会经常被修改，或者被切换。

通常情况下应该是被切换，因为过了促销时间，又还回到正常的价格体系上来了。而现在的价格类中计算报价的方法，是固定调用各种计算方式，这使得切换调用不同的计算方式很麻烦，每次都需要修改 if-else 中的调用代码。

看到这里，可能有朋友会想，那么到底应该如何实现，才能够让价格类中的计算报价的算法，能很容易地实现可维护、可扩展，又能动态地切换变化呢？

17.2 解决方案

17.2.1 使用策略模式来解决问题

用来解决上述问题的一个合理的解决方案就是策略模式。那么什么是策略模式呢？

1. 策略模式的定义

> 定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

2. 应用策略模式来解决问题的思路

仔细分析上面的问题，先来把它抽象一下，各种计算报价的计算方式就好比是具体的算法，而使用这些计算方式来计算报价的程序，就相当于是使用算法的客户。

再分析上面的实现方式，为什么会造成那些问题？根本原因就在于算法和使用算法的客户是耦合的，甚至是密不可分的。在上面的实现中，具体的算法和使用算法的客户是同一个类中的不同方法。

现在来解决那些问题。按照策略模式的方式，应该先把所有的计算方式独立出来，每个计算方式做成一个单独的算法类，从而形成一系列的算法，并且为这一系列算法定义一个公共的接口，这些算法实现是同一接口的不同实现，地位是平等的，可以相互替换。这样一来，要扩展新的算法就变成了增加一个新的算法实现类，要维护某个算法，也只是修改某个具体的算法实现即可，不会对其他代码造成影响。也就是说这样就解决了可维护、可扩展的问题。

为了实现让算法能独立于使用他的客户，策略模式引入了一个上下文的对象，这个

对象负责持有算法，但是不负责决定具体选用哪个算法，把选择算法的功能交给了客户，由客户选择好具体的算法后，设置到上下文对象中，让上下文对象持有客户选择的算法，当客户通知上下文对象执行功能的时候，上下文对象则转调具体的算法。这样一来，具体的算法和直接使用算法的客户是分离的。

具体的算法和使用他的客户分离以后，使得算法可独立于使用它的客户而变化，并且能够动态地切换需要使用的算法，只要客户端动态地选择使用不同的算法，然后设置到上下文对象中去，在实际调用的时候，就可以调用到不同的算法。

17.2.2 策略模式的结构和说明

策略模式的结构如图 17.1 所示。

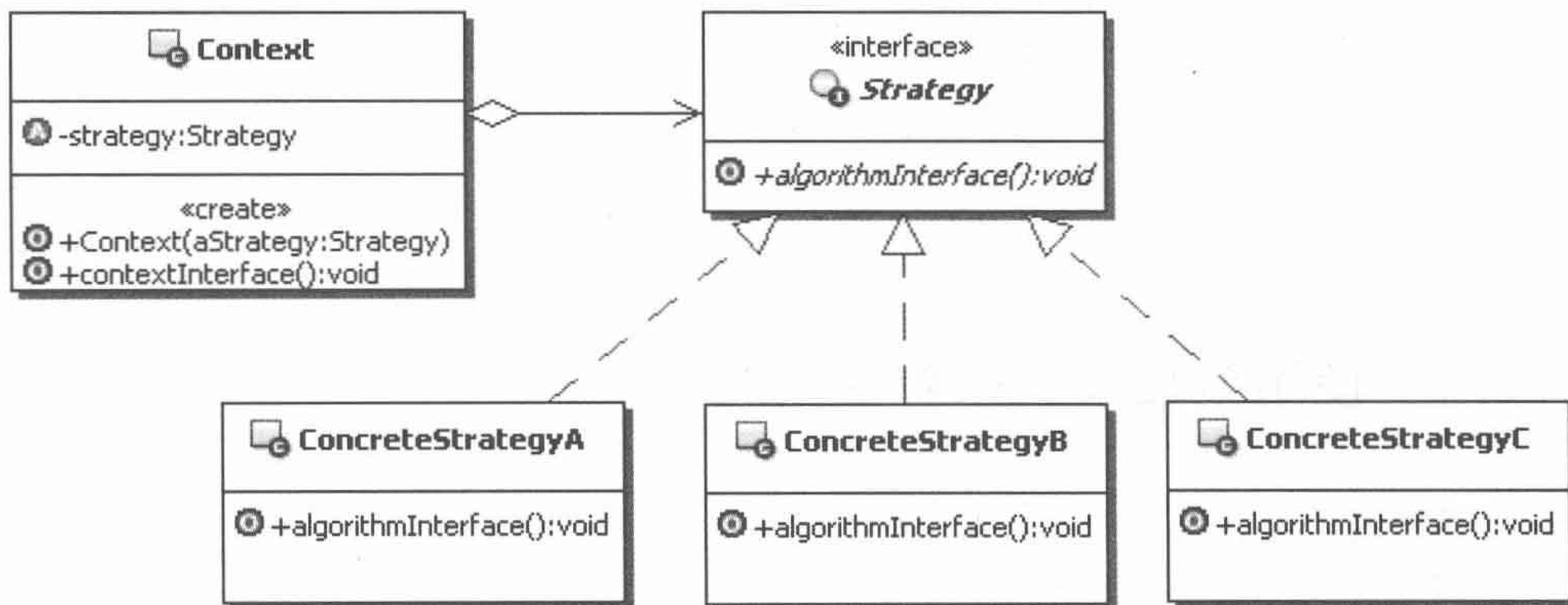


图 17.1 策略模式的结构示意图

- Strategy：策略接口，用来约束一系列具体的策略算法。Context 使用这个接口来调用具体的策略实现定义的算法。
- ConcreteStrategy：具体的策略实现，也就是具体的算法实现。
- Context：上下文，负责和具体的策略类交互。通常上下文会持有一个真正的策略实现，上下文还可以让具体的策略类来获取上下文的数据，甚至让具体的策略类来回调上下文的方法。

17.2.3 策略模式示例代码

（1）首先来看策略，也就是定义算法的接口，示例代码如下：

```
/**
 * 策略，定义算法的接口
 */
public interface Strategy {
    /**
     * 某个算法的接口，可以有传入参数，也可以有返回值
     */
    public void algorithmInterface();
```

```
}
```

（2）再来看看具体的算法实现。定义了三个算法，分别是 ConcreteStrategyA、ConcreteStrategyB、ConcreteStrategyC，示例非常简单，由于没有具体算法的实现，三者也就是名称不同。示例代码如下：

```
/**
 * 实现具体的算法
 */
public class ConcreteStrategyA implements Strategy {
    public void algorithmInterface() {
        //具体的算法实现
    }
}
/**
 * 实现具体的算法
 */
public class ConcreteStrategyB implements Strategy {
    public void algorithmInterface() {
        //具体的算法实现
    }
}
/**
 * 实现具体的算法
 */
public class ConcreteStrategyC implements Strategy {
    public void algorithmInterface() {
        //具体的算法实现
    }
}
```

（3）接下来看看上下文的实现。示例代码如下：

```
/**
 * 上下文对象，通常会持有一个具体的策略对象
 */
public class Context {
    /**
     * 持有一个具体的策略对象
     */
    private Strategy strategy;
    /**
     * 构造方法，传入一个具体的策略对象
```

```
     * @param aStrategy 具体的策略对象
     */
    public Context(Strategy aStrategy) {
        this.strategy = aStrategy;
    }
    /**
     * 上下文对客户端提供的操作接口，可以有参数和返回值
     */
    public void contextInterface() {
        //通常会转调具体的策略对象进行算法运算
        strategy.algorithmInterface();
    }
}
```

17.2.4 使用策略模式重写示例

要使用策略模式来重写前面报价的示例，大致有如下改变。

（1）首先需要定义出算法的接口。

（2）然后把各种报价的计算方式单独出来，形成算法类。

（3）对于 Price 类，把它当做上下文，在计算报价的时候，不再需要判断，直接使用持有的具体算法进行运算即可。具体选择使用哪一个算法的功能挪出去，放到外部使用的客户端去。

这个时候，程序的结构如图 17.2 所示。

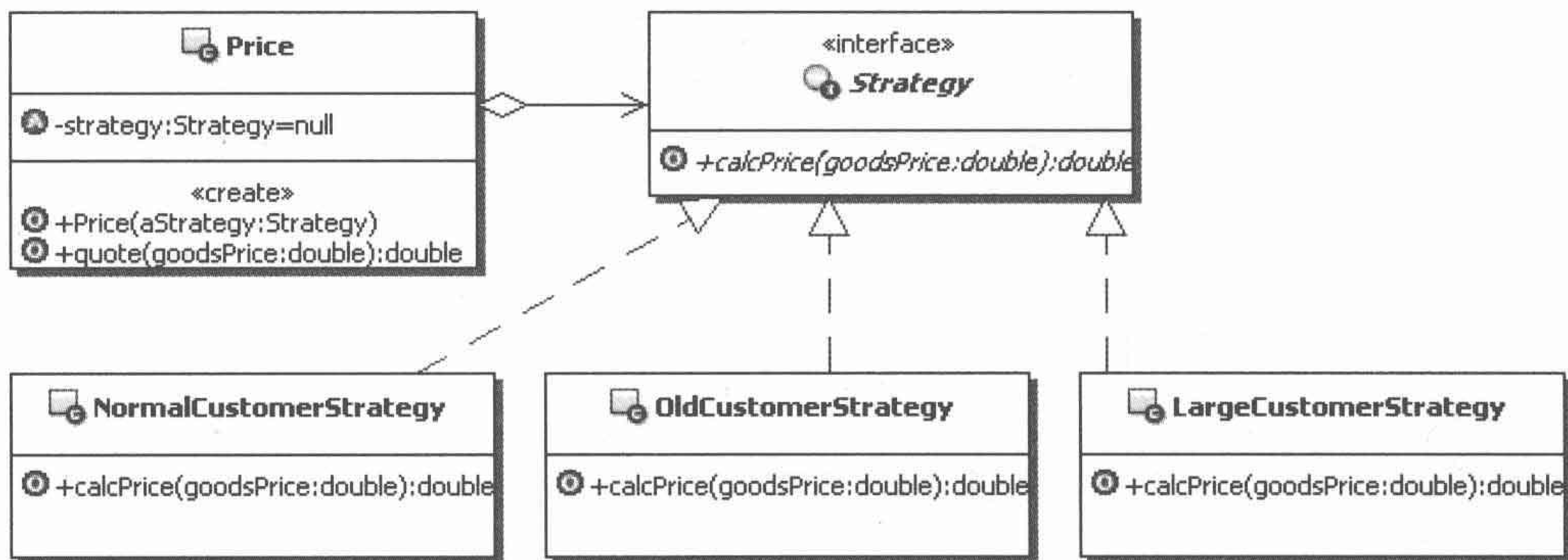


图 17.2　使用策略模式实现示例的结构示意图

（1）先来看策略接口。示例代码如下：

```
/**
 * 策略，定义计算报价算法的接口
 */
public interface Strategy {
    /**
```

```
     * 计算应报的价格
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    public double calcPrice(double goodsPrice);
}
```

（2）下面来看看具体的算法实现。不同的算法，实现也不一样。

先来看看为新客户或者是普通客户计算应报价格的实现。示例代码如下：

```
/**
 * 具体算法实现，为新客户或者是普通客户计算应报的价格
 */
public class NormalCustomerStrategy implements Strategy{
    public double calcPrice(double goodsPrice) {
        System.out.println("对于新客户或者是普通客户，没有折扣");
        return goodsPrice;
    }
}
```

再看看为老客户计算应报价格的实现。示例代码如下：

```
/**
 * 具体算法实现，为老客户计算应报的价格
 */
public class OldCustomerStrategy implements Strategy{
    public double calcPrice(double goodsPrice) {
        System.out.println("对于老客户，统一折扣5%");
        return goodsPrice*(1-0.05);
    }
}
```

接下来看看为大客户计算应报价格的实现。示例代码如下：

```
/**
 * 具体算法实现，为大客户计算应报的价格
 */
public class LargeCustomerStrategy implements Strategy{
    public double calcPrice(double goodsPrice) {
        System.out.println("对于大客户，统一折扣10%");
        return goodsPrice*(1-0.1);
    }
}
```

（3）下面来看看上下文的实现，也就是原来的价格类，它的变化比较大，主要有：

- 原来那些私有的，用来做不同计算的方法，已经删除了，独立出去做成了算法类。
- 原来报价方法中，对具体计算方式的判断，删除了，让客户端来完成选择具体算法的功能。
- 新添加持有一个具体的算法实现，通过构造方法传入。
- 原来报价方法的实现，变化成了转调具体算法来实现。

示例代码如下：

```
/**
 * 价格管理，主要完成计算向客户所报价格的功能
 */
public class Price {
    /**
     * 持有一个具体的策略对象
     */
    private Strategy strategy = null;
    /**
     * 构造方法，传入一个具体的策略对象
     * @param aStrategy 具体的策略对象
     */
    public Price(Strategy aStrategy){
        this.strategy = aStrategy;
    }
    /**
     * 报价，计算对客户的报价
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    public double quote(double goodsPrice){
        return this.strategy.calcPrice(goodsPrice);
    }
}
```

（4）写个客户端来测试运行一下，以加深体会。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //1：选择并创建需要使用的策略对象
        Strategy strategy = new LargeCustomerStrategy ();
        //2：创建上下文
        Price ctx = new Price(strategy);

        //3：计算报价
```

```
            double quote = ctx.quote(1000);
            System.out.println("向客户报价："+quote);
        }
}
```

运行一下，看看效果。

你可以修改使用不同的策略算法具体实现，现在用的是 LargeCustomerStrategy，你可以尝试修改成其他两种实现，试试看，体会一下切换算法的容易性。

17.3 模式讲解

17.3.1 认识策略模式

1. 策略模式的功能

策略模式的功能是把具体的算法实现从具体的业务处理中独立出来，把它们实现成为单独的算法类，从而形成一系列的算法，并让这些算法可以相互替换。

> 提示　**策略模式的重心不是如何来实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活，具有更好的维护性和扩展性。**

2. 策略模式和 if-else 语句

看了前面的示例，很多朋友会发现，每个策略算法具体实现的功能，就是原来在 if-else 结构中的具体实现。

没错，其实多个 if-elseif 语句表达的就是一个平等的功能结构，你要么执行 if，要么执行 else，或者是 elseif，这个时候，if 块中的实现和 else 块中的实现从运行地位上来讲是平等的。

而策略模式就是把各个平等的具体实现封装到单独的策略实现类了，然后通过上下文来与具体的策略类进行交互。

因此多个 if-else 语句可以考虑使用策略模式。

3. 算法的平等性

策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正是因为这个平等性，才能实现算法之间可以相互替换。

所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。

所以可以这样描述这一系列策略算法：**策略算法是相同行为的不同实现**。

4. 谁来选择具体的策略算法

在策略模式中，可以在两个地方来进行具体策略的选择。

一个是在客户端，当使用上下文的时候，由客户端来选择具体的策略算法，然后把

这个策略算法设置给上下文。前面的示例就是这种情况。

还有一个是客户端不管，由上下文来选择具体的策略算法，这个在后面介绍容错恢复的时候给大家演示一下。

5. Strategy 的实现方式

在前面的示例中，Strategy 都是使用接口来定义的，这也是常见的实现方式。但是如果多个算法具有公共功能的话，可以把 Strategy 实现成为抽象类，然后把多个算法的公共功能实现到 Strategy 中。

6. 运行时策略的唯一性

运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态地在不同的策略实现中切换，但是同时只能使用一个。

7. 增加新的策略

在前面的示例中，体会到了策略模式中切换算法的方便，但是增加一个新的算法会怎样呢？比如现在要实现如下的功能，对于公司的“战略合作客户”，统一 8 折。

其实很简单，策略模式可以让你很灵活地扩展新的算法。具体的做法是，先写一个策略算法类来实现新的要求，然后在客户端使用的时候指定使用新的策略算法类就可以了。

还是通过示例来说明。先添加一个实现要求的策略类。示例代码如下：

```
/**
 * 具体算法实现，为战略合作客户计算应报的价格
 */
public class CooperateCustomerStrategy implements Strategy{
    public double calcPrice(double goodsPrice) {
        System.out.println("对于战略合作客户，统一8折");
        return goodsPrice*0.8;
    }
}
```

然后在客户端指定使用策略的时候指定新的策略算法实现。示例代码如下：

```
public class Client2 {
    public static void main(String[] args) {
        //1：选择并创建需要使用的策略对象
        Strategy strategy = new CooperateCustomerStrategy ();
        //2：创建上下文
        Price ctx = new Price(strategy);

        //3：计算报价
        double quote = ctx.quote(1000);
        System.out.println("向客户报价："+quote);
    }
```

除了这里变动外，客户端没有其他的变化

```
}
```

运行客户端，测试看看，好好体会一下。

除了客户端发生变化外，已有的上下文、策略接口定义和策略的已有实现，都不需要做任何的修改，可见能很方便地扩展新的策略算法。

8. 策略模式的调用顺序示意图

策略模式的调用顺序，有两种常见的情况，一种如同前面的示例，具体如下：

（1）先是客户端来选择并创建具体的策略对象。

（2）然后客户端创建上下文。

（3）接下来客户端就可以调用上下文的方法来执行功能了，在调用的时候，从客户端传入算法需要的参数。

（4）上下文接到客户的调用请求，会把这个请求转发给它持有的 Strategy。

这种情况的调用顺序如图 17.3 所示。



图 17.3　策略模式调用顺序示意图一

策略模式的调用还有一种情况，就是把 Context 当作参数来传递给 Strategy，这种方式的调用顺序图，在介绍具体的 Context 和 Strategy 的关系时再给出。

17.3.2　Context 和 Strategy 的关系

在策略模式中，通常是上下文使用具体的策略实现对象。反过来，策略实现对象也可以从上下文获取所需要的数据。因此可以将上下文当作参数传递给策略实现对象，这种情况下上下文和策略实现对象是紧密耦合的。

在这种情况下，上下文封装着具体策略对象进行算法运算所需要的数据，具体策略对象通过回调上下文的方法来获取这些数据。

甚至在某些情况下，策略实现对象还可以回调上下文的方法来实现一定的功能，这种使用场景下，上下文变相充当了多个策略算法实现的公共接口。在上下文定义的方法可以当作是所有或者是部分策略算法使用的公共功能。

> 注意　**但是请注意，由于所有的策略实现对象都实现同一个策略接口，传入同一个上下文，可能会造成传入的上下文数据的浪费，因为有的算法会使用这些数据，而有的算法不会使用，但是上下文和策略对象之间交互的开销是存在的。**

还是通过例子来说明。

1. 工资支付的实现思路

考虑这样一个功能：工资支付方式的问题。很多企业的工资支付方式是很灵活的，可支付方式是比较多的，比如，人民币现金支付、美元现金支付、银行转账到工资账户、银行转账到工资卡；一些创业型的企业为了留住骨干员工，还可能有工资转股权等方式。总之一句话，工资支付方式很多。

随着公司的发展，会不断有新的工资支付方式出现，这就要求能方便地扩展；另外工资支付方式不是固定的，是由公司和员工协商确定的，也就是说可能不同的员工采用的是不同的支付方式，甚至同一个员工，不同时间采用的支付方式也可能会不同，这就要求能很方便地切换具体的支付方式。

要实现这样的功能，策略模式是一个很好的选择。在实现这个功能的时候，不同的策略算法需要的数据是不一样，比如，现金支付就不需要银行账号，而银行转账就需要账号。这就导致在设计策略接口中的方法时，不太好确定参数的个数，而且，就算现在把所有的参数都列上了，今后扩展呢？难道再来修改策略接口吗？如果这样做，那无异于一场灾难，加入一个新策略，就需要修改接口，然后修改所有已有的实现，不疯掉才怪！那么到底如何实现，在今后扩展的时候才最方便呢？

解决方案之一，就是把上下文当作参数传递给策略对象。这样一来，如果要扩展新的策略实现，只需要扩展上下文就可以了，已有的实现不需要做任何的修改。

这样是不是能很好地实现功能，并具有很好的扩展性呢？还是通过代码示例来具体看看。假设先实现人民币现金支付和美元现金支付这两种支付方式，然后进行使用测试，再来添加银行转账到工资卡的支付方式，看看是不是能很容易的与已有地实现结合上。

2. 实现代码示例

（1）先定义工资支付的策略接口，也就是定义一个支付工资的方法。示例代码如下：

```
/**
 * 支付工资的策略接口，公司有多种支付工资的算法
 * 比如，现金、银行卡、现金加股票、现金加期权、美元支付等
 */
public interface PaymentStrategy {
    /**
     * 公司给某人真正支付工资
     * @param ctx 支付工资的上下文，里面包含算法需要的数据
     */
    public void pay(PaymentContext ctx);
}
```

（2）定义好了工资支付的策略接口，该来考虑如何实现这多种支付策略了。

为了演示的简单，这里先简单实现人民币现金支付和美元现金支付方式，当然并不是真地去实现跟银行的交互，只是示意一下。

人民币现金支付的策略实现。示例代码如下：

```
/**
 * 人民币现金支付
 */
public class RMBCash implements PaymentStrategy{
    public void pay(PaymentContext ctx) {
        System.out.println("现在给"+ctx.getUserName()
                              +"人民币现金支付"+ctx.getMoney()+"元");
    }
}
```

同样地实现美元现金支付的策略。示例代码如下：

```
/**
 * 美元现金支付
 */
public class DollarCash implements PaymentStrategy{
    public void pay(PaymentContext ctx) {
        System.out.println("现在给"+ctx.getUserName()
                              +"美元现金支付"+ctx.getMoney()+"元");
    }
}
```

（3）该来看支付上下文的实现了，当然这个使用支付策略的上下文，是需要知道具体使用哪一个支付策略的，一般由客户端来确定具体使用哪一个具体的策略，然后上下文负责去真正执行。因此，这个上下文需要持有一个支付策略，而且是由客户端来配置它。示例代码如下：

```
/**
 * 支付工资的上下文，每个人的工资不同，支付方式也不同
 */
public class PaymentContext {
    /**
     * 应被支付工资的人员，简单点，用姓名来代替
     */
    private String userName = null;
    /**
     * 应被支付的工资金额
     */
    private double money = 0.0;
    /**
     * 支付工资的方式的策略接口
     */
    private PaymentStrategy strategy = null;
```

```
    /**
     * 构造方法，传入被支付工资的人员，应支付的金额和具体的支付策略
     * @param userName 被支付工资的人员
     * @param money 应支付的金额
     * @param strategy 具体的支付策略
     */
    public PaymentContext(String userName,double money,
                                    PaymentStrategy strategy){
        this.userName = userName;
        this.money = money;
        this.strategy = strategy;
    }
    public String getUserName() {
        return userName;
    }
    public double getMoney() {
        return money;
    }
   /**
     * 立即支付工资
     */
    public void payNow(){
        //使用客户希望的支付策略来支付工资
        this.strategy.pay(this);
    }
}
```

只有 getter 方法，让策略算法在实现的时候，根据需要来获取上下文中的数据

(4)准备好了支付工资的各种策略，下面来看看如何使用这些策略来真正支付工资。很简单，客户端是使用上下文来使用具体的策略的，而且是客户端来确定具体的策略，就是客户端创建哪个策略，最终就运行哪一个策略，各个策略之间是可以动态切换的。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建相应的支付策略
        PaymentStrategy strategyRMB = new RMBCash();
        PaymentStrategy strategyDollar = new DollarCash();

        //准备小李的支付工资上下文
        PaymentContext ctx1 =
                new PaymentContext("小李",5000,strategyRMB);
        //向小李支付工资
```

```
        ctx1.payNow();

        //切换一个人，给petter支付工资
        PaymentContext ctx2 =
                new PaymentContext("Petter",8000,strategyDollar);
        ctx2.payNow();
    }
}
```

运行一下，看看效果。运行结果如下：

```
现在给小李人民币现金支付5000.0元
现在给Petter美元现金支付8000.0元
```

3. 扩展示例，实现方式一

经过上面的测试可以看出，通过使用策略模式，已经实现好了两种支付方式了。如果现在要增加一种支付方式，要求能支付到银行卡，该怎样扩展最简单呢？

应该新增加一种支付到银行卡的策略实现，然后通过继承来扩展支付上下文，在其中添加新的支付方式需要的新数据，比如银行卡账户，并在客户端使用新的上下文和新的策略实现就可以了，这样已有的实现都不需要改变，完全遵循开—闭原则。

先看看扩展的支付上下文对象的实现。示例代码如下：

```
/**
 * 扩展的支付上下文对象
 */
public class PaymentContext2 extends PaymentContext {
    /**
     * 银行账号
     */
    private String account = null;
    /**
     * 构造方法，传入被支付工资的人员，应支付的金额和具体的支付策略
     * @param userName 被支付工资的人员
     * @param money 应支付的金额
     * @param account 支付到的银行账号
     * @param strategy 具体的支付策略
     */
    public PaymentContext2(String userName,double money,
                    String account,PaymentStrategy strategy){
        super(userName,money,strategy);
        this.account = account;
    }
    public String getAccount() {
```

```
        return account;
    }
}
```

然后看看新的策略算法的实现。示例代码如下：

```
/**
 * 支付到银行卡
 */
public class Card implements PaymentStrategy{
    public void pay(PaymentContext ctx) {
        //这个新的算法自己知道要使用扩展的支付上下文，所以强制造型一下
        PaymentContext2 ctx2 = (PaymentContext2)ctx;
        System.out.println("现在给"+ctx2.getUserName()+"的"
            +ctx2.getAccount()+"账号支付了"+ctx2.getMoney()+"元");
        //连接银行，进行转账，就不去管了
    }
}
```

最后看看客户端怎么使用这个新的策略呢？原有的代码不变，直接添加新的测试就可以了。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建相应的支付策略
        PaymentStrategy strategyRMB = new RMBCash();
        PaymentStrategy strategyDollar = new DollarCash();

        //准备小李的支付工资上下文
        PaymentContext ctx1 =
                new PaymentContext("小李",5000,strategyRMB);
        //向小李支付工资
        ctx1.payNow();

        //切换一个人，给petter支付工资
        PaymentContext ctx2 =
                new PaymentContext("Petter",8000,strategyDollar);
        ctx2.payNow();

        //测试新添加的支付方式
        PaymentStrategy strategyCard = new Card();
        PaymentContext ctx3 = new PaymentContext2(
                    "小王",9000,"010998877656",strategyCard);
```

```
        ctx3.payNow();
    }
}
```

再次测试，体会一下。运行结果如下：

```
现在给小李人民币现金支付5000.0元
现在给Petter美元现金支付8000.0元
现在给小王的010998877656账号支付了9000.0元
```

新加的策略测试结果

4. 扩展示例，实现方式二

同样还是实现上面这个功能：现在要增加一种支付方式，要求能支付到银行卡。

（1）上面这种实现方式，是通过扩展上下文对象来准备新的算法需要的数据。还有另外一种方式，那就是通过策略的构造方法来传入新算法需要的数据。这样实现的话，就不需要扩展上下文了，直接添加新的策略算法实现就可以了。示例代码如下：

```
/**
 * 支付到银行卡
 */
public class Card2 implements PaymentStrategy{
    /**
     * 账号信息
     */
    private String account = "";
    /**
     * 构造方法，传入账号信息
     * @param account 账号信息
     */
    public Card2(String account){
        this.account = account;
    }
    public void pay(PaymentContext ctx) {
        System.out.println("现在给"+ctx.getUserName()+"的"
                +this.account+"账号支付了"+ctx.getMoney()+"元");
        //连接银行，进行转账，就不去管了
    }
}
```

（2）直接在客户端测试就可以了。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //测试新添加的支付方式
        PaymentStrategy strategyCard2 = new Card2("010998877656");
```

```
        PaymentContext ctx4 =
                new PaymentContext("小张",9000,strategyCard2);
        ctx4.payNow();
    }
}
```

运行看看，好好体会一下。

（3）现在有这么两种扩展的实现方式，到底使用哪一种呢？或者是哪种实现更好呢？下面来比较一下。

- **对于扩展上下文的方式**：这样实现，所有策略的实现风格更统一，策略需要的数据都统一从上下文来获取，这样在使用方法上也很统一；另外，在上下文中添加新的数据，别的相应算法也可以用得上，可以视为公共的数据。但缺点也很明显，如果这些数据只有一个特定的算法来使用，那么这些数据有些浪费；另外每次添加新的算法都去扩展上下文，容易形成复杂的上下文对象层次，也未见得有必要。
- **对于在策略算法的实现上添加自己需要的数据的方式**：这样实现，比较好想，实现起来简单。但是缺点也很明显，跟其他策略实现的风格不一致，其他策略都是从上下文中来获取数据，而这个策略的实现一部分数据来自上下文，一部分数据来自自己，有些不统一；另外，这样一来，外部使用这些策略算法的时候也不一样了，难于以一个统一的方式来动态切换策略算法。

两种实现各有优劣，至于如何选择，那就具体问题具体分析了。

5. 另一种策略模式调用顺序示意图

策略模式调用还有一种情况，就是把 Context 当作参数来传递给 Strategy，也就是本例示范的这种方式，这个时候策略模式的调用顺序如图 17.4 所示。

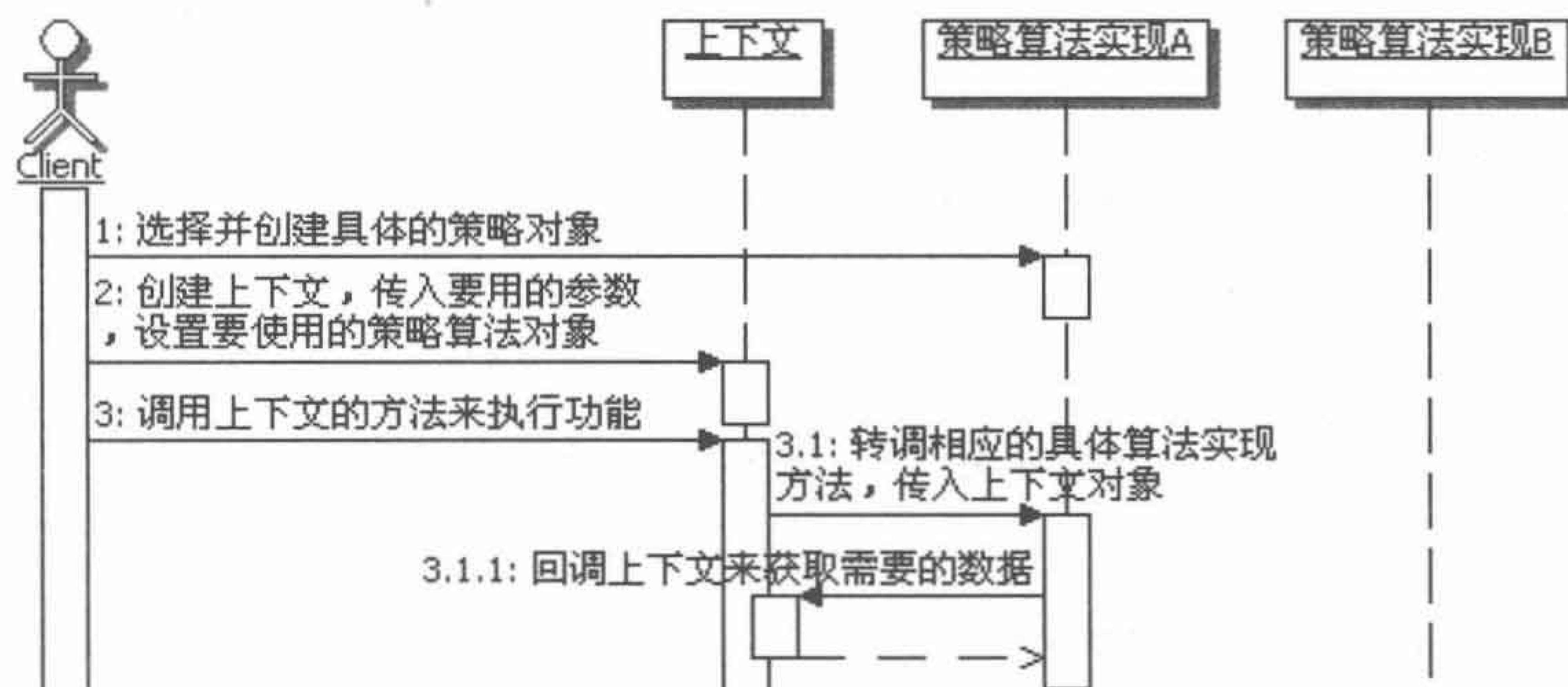


图 17.4　策略模式调用顺序示意图二

17.3.3　容错恢复机制

容错恢复机制是应用程序开发中常见的功能。那么什么是容错恢复呢？简单点说就是，程序运行的时候，正常情况下应该按照某种方式来做，如果按照某种方式来做发生错误的话，系统并不会崩溃，也不会就此不能继续向下运行了，而是有容忍出错的能力，不但能容忍程序运行出现错误，还提供出现错误后的备用方案，也就是恢复机制，来代替正常执行的功能，使程序继续向下运行。

举个实际点的例子吧，比如在一个系统中，所有对系统的操作都要有日志记录，而且这个日志还需要有管理界面，这种情况下通常会把日志记录在数据库里面，方便后续的管理，但是在记录日志到数据库的时候，可能会发生错误，比如暂时连不上数据库了，那就先记录在文件里面，然后在合适的时候把文件中的记录再转录到数据库中。

对于这样的功能的设计，就可以采用策略模式，把日志记录到数据库和日志记录到文件当作两种记录日志的策略，然后在运行期间根据需要进行动态的切换。

在这个例子的实现中，要示范由上下文来选择具体的策略算法，而前面的例子都是由客户端选择好具体的算法，然后设置到上下文中。

下面还是通过代码来示例一下。

（1）先定义日志策略接口，很简单，就是一个记录日志的方法。示例代码如下：

```
/**
 * 日志记录策略的接口
 */
public interface LogStrategy {
    /**
     * 记录日志
     * @param msg 需记录的日志信息
     */
    public void log(String msg);
}
```

（2）实现日志策略接口。

先实现默认的数据库实现，假设如果日志的长度超过长度就出错，制造错误的是一个最常见的运行期错误。示例代码如下：

```
/**
 * 把日志记录到数据库
 */
public class DbLog implements LogStrategy{
    public void log(String msg) {
        //制造错误
        if(msg!=null && msg.trim().length()>5){
            int a = 5/0;
        }
        System.out.println("现在把 '"+msg+"' 记录到数据库中");
    }
}
```

接下来实现记录日志到文件中去。示例代码如下：

```
/**
 * 把日志记录到文件
 */
public class FileLog implements LogStrategy{
    public void log(String msg) {
        System.out.println("现在把 '"+msg+"' 记录到文件中");
    }
}
```

（3）下面来定义使用这些策略的上下文。注意这次是在上下文中实现具体策略算法的选择，所以不需要客户端来指定具体的策略算法了。示例代码如下：

```
/**
 * 日志记录的上下文
 */
public class LogContext {
    /**
     * 记录日志的方法，提供给客户端使用
     * @param msg 需记录的日志信息
     */
    public void log(String msg){
        //在上下文中，自行实现对具体策略的选择
        //优先选用策略：记录到数据库
        LogStrategy strategy = new DbLog();
        try{
            strategy.log(msg);
        }catch(Exception err){
            //出错了，那就记录到文件中
            strategy = new FileLog();
            strategy.log(msg);
        }
    }
}
```

在这里进行具体策略算法的选择，把 try-catch 变相当成了 if-else 来用

（4）看看现在的客户端，没有了选择具体实现策略算法的工作，变得非常简单。故意多调用一次，可以看出不同的效果。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        LogContext log = new LogContext();
        log.log("记录日志");
        log.log("再次记录日志");
    }
```

看起来这两句是没有什么区别的吧，运行一下看看结果，看看会发生什么

```
}
```

运行结果如下：

```
现在把 '记录日志' 记录到数据库中
现在把 '再次记录日志' 记录到文件中
```

为什么运行的结果是一个记录到了数据库中，一个记录到了文件中呢？很简单，第二次调用记录日志的日志消息超长了，运行出错，容错恢复，所以记录日志到文件中去

（5）小结一下。通过上面的示例，看到策略模式的一种简单应用，也顺便了解了基本容错恢复机制的设计和实现。在实际的应用中，需要设计容错恢复的系统一般要求都比较高，应用也会更加复杂，但是基本的思路是差不多的。

17.3.4　策略模式结合模板方法模式

在实际应用策略模式的过程中，经常会出现这样一种情况，就是发现这一系列算法的实现上存在公共功能，甚至这一系列算法的实现步骤都是一样的，只是在某些局部步骤上有所不同，这个时候，就需要对策略模式进行些许的变化使用了。

对于一系列算法的实现上存在公共功能的情况，策略模式可以有以下三种实现方式。

- 在上下文当中实现公共功能，让所有具体的策略算法回调这些方法。
- 将策略的接口改成抽象类，然后在其中实现具体算法的公共功能。
- 为所有的策略算法定义一个抽象的父类，让这个父类去实现策略的接口，然后在这个父类中去实现公共的功能。

更进一步，如果这个时候发现“一系列算法的实现步骤都是一样的，只是在某些局部步骤上有所不同”的情况，那就可以在这个抽象类里面定义算法实现的骨架，然后让具体的策略算法去实现变化的部分。这样的一个结构自然就变成了策略模式结合模板方法模式了，那个抽象类就成了模板方法模式的模板类。

在第 16 章我们讨论过模板方法模式结合策略模式的方式，也就是主要的结构是模板方法模式，局部采用策略模式。而这里讨论的是策略模式结合模板方法模式，也就是主要的结构是策略模式，局部实现上采用模板方法模式。**通过这个示例也可以看出，模式之间的结合是没有定势的，要具体问题具体分析。**

此时策略模式结合模板方法模式的系统结构如图 17.5 所示。

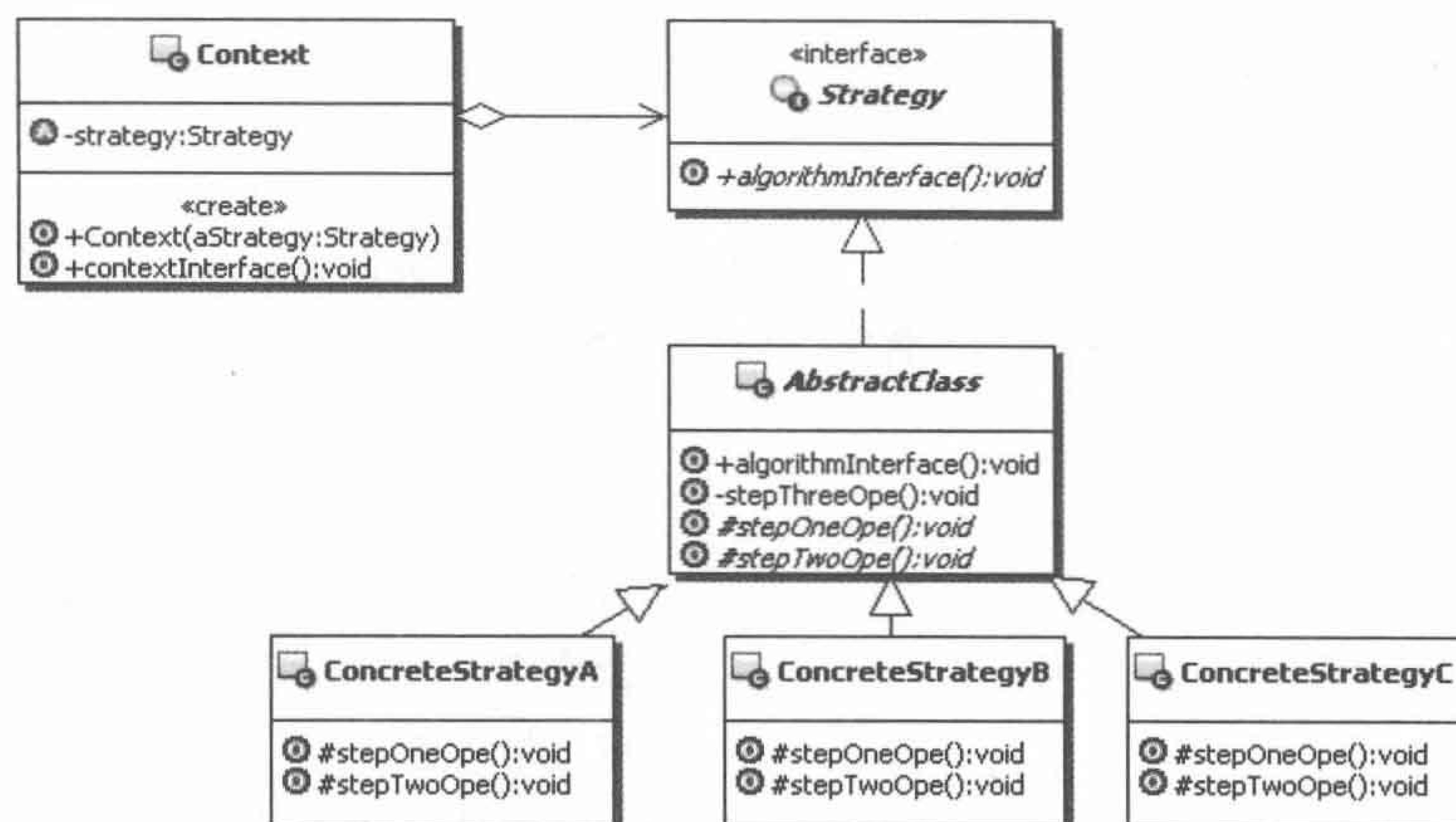


图 17.5 策略模式结合模板方法模式的结构示意图

还是用实际的例子来说吧，比如上面那个记录日志的例子，如果现在需要在所有的消息前面都添加上日志时间，也就是说现在记录日志的步骤变成了：第一步为日志消息添加日志时间；第二步具体记录日志。

那么该怎么实现呢？

（1）记录日志的策略接口没有变化，为了看起来方便，还是示例一下。示例代码如下：

```
/**
 * 日志记录策略的接口
 */
public interface LogStrategy {
    /**
     * 记录日志
     * @param msg 需记录的日志信息
     */
    public void log(String msg);
}
```

（2）增加一个实现这个策略接口的抽象类，在其中定义记录日志的算法骨架，相当于模板方法模式的模板。示例代码如下：

```
/**
 * 实现日志策略的抽象模板，实现为消息添加时间
 */
public abstract class LogStrategyTemplate implements LogStrategy{
    public final void log(String msg) {
        //第一步：为消息添加记录日志的时间
        DateFormat df = new SimpleDateFormat(
                                    "yyyy-MM-dd HH:mm:ss SSS");
        msg = df.format(new java.util.Date())+" 内容是："+ msg;
```

```
        //第二步：真正执行日志记录
        doLog(msg);
    }
    /**
     * 真正执行日志记录，让子类去具体实现
     * @param msg 需记录的日志信息
     */
    protected abstract void doLog(String msg);
}
```

（3）这个时候那两个具体的日志算法实现也需要做些改变，不再直接实现策略接口了，而是继承模板，实现模板方法。这个时候记录日志到数据库的类。示例代码如下：

```
/**
 * 把日志记录到数据库
 */
public class DbLog extends LogStrategyTemplate{
    public void doLog(String msg) {
        //制造错误
        if(msg!=null && msg.trim().length()>5){
            int a = 5/0;
        }
        System.out.println("现在把 '"+msg+"' 记录到数据库中");
    }
}
```

除了定义上发生了改变外，具体的实现没变

同理实现记录日志到文件的类如下：

```
/**
 * 把日志记录到数据库
 */
public class FileLog extends LogStrategyTemplate{
    public void doLog(String msg) {
        System.out.println("现在把 '"+msg+"' 记录到文件中");
    }
}
```

（4）算法实现的改变不影响使用算法的上下文，上下文和前面一样。示例代码如下：

```
/**
 * 日志记录的上下文
 */
public class LogContext {
    /**
     * 记录日志的方法，提供给客户端使用
```

```
     * @param msg 需记录的日志信息
     */
    public void log(String msg){
        //在上下文中，自行实现对具体策略的选择
        //优先选用策略：记录到数据库
        LogStrategy strategy = new DbLog();
        try{
            strategy.log(msg);
        }catch(Exception err){
            //出错了，那就记录到文件中
            strategy = new FileLog();
            strategy.log(msg);
        }
    }
}
```

（5）客户端和以前也一样。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        LogContext log = new LogContext();
        log.log("记录日志");
        log.log("再次记录日志");
    }
}
```

运行一下客户端再次测试看看，体会一下，看看结果是否带上了时间。

通过这个示例，好好体会一下策略模式和模板方法模式的组合使用，这在实用开发中是很常见的方式。

17.3.5 策略模式的优缺点

策略模式有以下优点。

- 定义一系列算法

  策略模式的功能就是定义一系列算法，实现让这些算法可以相互替换。所以会为这一系列算法定义公共的接口，以约束一系列算法要实现的功能。如果这一系列算法具有公共功能，可以把策略接口实现成为抽象类，把这些公共功能实现到父类中，对于这个问题，前面讲了三种处理方法，这里就不再啰嗦了。

- 避免多重条件语句

  根据前面的示例会发现，策略模式的一系列策略算法是平等的，是可以互换的，写在一起就是通过 if-else 结构来组织，如果此时具体的算法实现中又有条件语句，就构成了多重条件语句，使用策略模式能避免这样的多重条件语句。

下面的示例演示了不使用策略模式的多重条件语句。示例代码如下：

```
public class OneClass {
    /**
     * 示范多重条件语句
     * @param type 某个用于判断的类型
     */
    public void oneMethod(int type){
        if(type==1){
            //算法一示范
            //从某个地方获取这个s的值
            String s = "";
            //然后判断进行相应处理
            if(s.indexOf("a") > 0){
                //处理
            }else{
                //处理
            }
        }else if(type==2){
            //算法二示范
            //从某个地方获取这个a的值
            int a = 3;
            //然后判断进行相应处理
            if(a > 10){
                //处理
            }else{
                //处理
            }
        }
    }
}
```

使用策略模式的时候，这些算法的处理代码就被拿出去，放到单独的算法实现类去了，这里就不再是多重条件了

- 更好的扩展性

  在策略模式中扩展新的策略实现非常容易，只要增加新的策略实现类，然后在使用策略的地方选择使用这个新的策略实现就可以了。

策略模式有以下缺点。

- 客户必须了解每种策略的不同

  策略模式也有缺点，比如让客户端来选择具体使用哪一个策略，这就需要客户了解所有的策略，还要了解各种策略的功能和不同，这样才能做出正确的选择，而且这样也暴露了策略的具体实现。

- 增加了对象数目

由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。

- 只适合扁平的算法结构

  策略模式的一系列算法地位是平等的，是可以相互替换的，事实上构成了一个扁平的算法结构，也就是在一个策略接口下，有多个平等的策略算法，就相当于兄弟算法。而且在运行时刻只有一个算法被使用，这就限制了算法使用的层级，使用的时候不能嵌套使用。

  对于出现需要嵌套使用多个算法的情况，比如折上折、折后返卷等业务的实现，需要组合或者是嵌套使用多个算法的情况，可以考虑使用装饰模式，或是变形的职责链，或是 AOP 等方式来实现。

17.3.6 思考策略模式

1. 策略模式的本质

> 策略模式的本质：**分离算法，选择实现**。

仔细思考策略模式的结构和实现的功能，会发现，如果没有上下文，策略模式就回到了最基本的接口和实现了，只要是面向接口编程的，那么就能够享受到接口的封装隔离带来的好处。也就是通过一个统一的策略接口来封装和隔离具体的策略算法，面向接口编程的话，自然不需要关心具体的策略实现，也可以通过使用不同的实现类来实例化接口，从而实现切换具体的策略。

看起来好像没有上下文什么事情，但是如果没有上下文，那么就需要客户端来直接与具体的策略交互，尤其是当需要提供一些公共功能，或者是相关状态存储的时候，会大大增加客户端使用的难度。因此，引入上下文还是很必要的，有了上下文，这些工作就由上下文来完成了，客户端只需要与上下文交互就可以了，这样会让整个设计模式更独立、更有整体性，也让客户端更简单。

但纵观整个策略模式实现的功能和设计，它的本质还是“分离算法，选择实现”，因为分离并封装了算法，才能够很容易地修改和添加算法；也能很容易地动态切换使用不同的算法，也就是动态选择一个算法来实现需要的功能。

2. 对设计原则的体现

从设计原则上来看，策略模式很好地体现了开—闭原则。策略模式通过把一系列可变的算法进行封装，并定义出合理的使用结构，使得在系统出现新算法的时候，能很容易地把新的算法加入到已有的系统中，而已有的实现不需要做任何修改。这在前面的示例中已经体现出来了，好好体会一下。

从设计原则上来看，策略模式还很好地体现了里氏替换原则。策略模式是一个扁平结构，一系列的实现算法其实是兄弟关系，都是实现同一个接口或者继承的同一个父类。这样只要使用策略的客户保持面向抽象类型编程，就能够使用不同策略的具体实现对象

来配置它，从而实现一系列算法可以相互替换。

3. 何时选用策略模式

建议在以下情况中选用策略模式。

- 出现有许多相关的类，仅仅是行为有差别的情况下，可以使用策略模式来使用多个行为中的一个来配置一个类的方法，实现算法动态切换。
- 出现同一个算法，有很多不同实现的情况下，可以使用策略模式来把这些“不同的实现”实现成为一个算法的类层次。
- 需要封装算法中，有与算法相关数据的情况下，可以使用策略模式来避免暴露这些跟算法相关的数据结构。
- 出现抽象一个定义了很多行为的类，并且是通过多个 if-else 语句来选择这些行为的情况下，可以使用策略模式来代替这些条件语句。

17.3.7　相关模式

- 策略模式和状态模式

  这两个模式从模式结构上看是一样的，但是实现的功能却是不一样的。

  状态模式是根据状态的变化来选择相应的行为，不同的状态对应不同的类，每个状态对应的类实现了该状态对应的功能，在实现功能的同时，还会维护状态数据的变化。这些实现状态对应的功能的类之间是不能相互替换的。策略模式是根据需要或者是客户端的要求来选择相应的实现类，各个实现类是平等的，是可以相互替换的。另外策略模式可以让客户端来选择需要使用的策略算法；而状态模式一般是由上下文，或者是在状态实现类里面来维护具体的状态数据，通常不由客户端来指定状态。
- 策略模式和模板方法模式

  这两个模式可组合使用，如同前面示例的那样。

  模板方法重在封装算法骨架；而策略模式重在分离并封装算法实现。
- 策略模式和享元模式

  这两个模式可组合使用。

  策略模式分离并封装出一系列的策略算法对象，这些对象的功能通常都比较单一，很多时候就是为了实现某个算法的功能而存在。因此，针对这一系列的、多个细粒度的对象，可以应用享元模式来节省资源，但前提是这些算法对象要被频繁地使用，如果偶尔用一次，就没有必要做成享元了。

读书笔记

第 18 章 状态模式（State）

18.1 场景问题

18.1.1 实现在线投票

考虑一个在线投票的应用，要实现控制同一个用户只能投一票，如果一个用户反复投票，而且投票次数超过 5 次，则判定为恶意刷票，要取消该用户投票的资格，当然同时也要取消他所投的票；如果一个用户的投票次数超过 8 次，将进入黑名单，禁止再登录和使用系统。

该怎么实现这样的功能呢？

18.1.2 不用模式的解决方案

分析上面的功能，为了控制用户投票，需要记录用户所投票的记录，同时还要记录用户投票的次数，为了简单，直接使用两个 Map 来记录。

在投票的过程中，又有四种情况：

- 用户是正常投票。
- 用户正常投票以后，有意或者无意地重复投票。
- 用户恶意投票。
- 黑名单用户。

这几种情况下对应的处理是不一样的。看看代码吧。示例代码如下：

```
/**
 * 投票管理
 */
public class VoteManager {
    /**
     * 记录用户投票的结果，Map<String,String>对应Map<用户名称,投票的选项>
     */
    private Map<String,String> mapVote =
                                   new HashMap<String,String>();
    /**
     * 记录用户投票次数，Map<String,Integer>对应Map<用户名称,投票的次数>
     */
    private Map<String,Integer> mapVoteCount =
                                 new HashMap<String,Integer>();
    /**
     * 投票
     * @param user 投票人，为了简单，就是用户名称
     * @param voteItem 投票的选项
```

```
     */
    public void vote(String user,String voteItem){
        //1：先为该用户增加投票的次数
        //从记录中取出已有的投票次数
        Integer oldVoteCount = mapVoteCount.get(user);
        if(oldVoteCount==null){
            oldVoteCount = 0;
        }
        oldVoteCount = oldVoteCount + 1;
        mapVoteCount.put(user, oldVoteCount);

        //2：判断该用户投票的类型，到底是正常投票、重复投票、恶意投票
       //还是上黑名单，然后根据投票类型来进行相应的操作
        if(oldVoteCount==1){
            //正常投票
            //记录到投票记录中
            mapVote.put(user, voteItem);
            System.out.println("恭喜你投票成功");
        }else if(oldVoteCount>1 && oldVoteCount<5){
            //重复投票
            //暂时不做处理
            System.out.println("请不要重复投票");
        }else if(oldVoteCount >= 5 && oldVoteCount<8){
            //恶意投票
            //取消用户的投票资格，并取消投票记录
            String s = mapVote.get(user);
            if(s!=null){
                mapVote.remove(user);
            }
            System.out.println("你有恶意刷票行为，取消投票资格");
        }else if(oldVoteCount>=8){
            //黑名单
            //记入黑名单中，禁止登录系统了
            System.out.println("进入黑名单，将禁止登录和使用本系统");
        }
    }
}
```

写个客户端来测试看看，是否能满足功能要求。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
```

```
        VoteManager vm = new VoteManager();
        for(int i=0;i<8;i++){
            vm.vote("u1", "A");
        }
    }
}
```

运行结果如下：

```
恭喜你投票成功
请不要重复投票
请不要重复投票
请不要重复投票
你有恶意刷票行为，取消投票资格
你有恶意刷票行为，取消投票资格
你有恶意刷票行为，取消投票资格
进入黑名单，将禁止登录和使用本系统
```

18.1.3 有何问题

看起来很简单，是不是？幸亏这里只是示意，否则，你想想，在 vote()方法中那么多判断，还有每个判断对应的功能处理都放在一起，是不是有点太杂乱了，那简直就是个大杂烩，如果把每个功能都完整地实现出来，那 vote()方法会很长的。

一个问题是，如果现在要修改某种投票情况所对应的具体功能处理，那就需要在那个大杂烩中，找到相应的代码块，然后进行改动。

另外一个问题是，如果要添加新的功能，比如投票超过 8 次但不足 10 次的，给个机会，只是禁止登录和使用系统 3 天，如果再犯，才永久封掉账号，该怎么办呢？那就需要改动投票管理的源代码，在上面的 if-else 结构中再添加一个 else if 块进行处理。

不管哪一种情况，都是在一大堆的控制代码中找出需要的部分，然后进行修改，这不是个好方法。那么该如何实现才能做到：既能够很容易地给 vote()方法添加新的功能，又能够很方便地修改已有的功能处理呢？

18.2 解决方案

18.2.1 使用状态模式来解决问题

用来解决上述问题的一个合理的解决方案就是状态模式。那么什么是状态模式呢？

1. 状态模式的定义

> 允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

2. 应用状态模式来解决的思路

仔细分析上面的问题，会发现，那几种用户投票的类型，就相当于是描述了人员的几种投票状态，而各个状态和对应的功能处理具有很强的对应性，有点类似于“一个萝卜一个坑”，各个状态下的处理基本上都是不一样的，也不存在可以相互替换的可能。

为了解决上面提出的问题，很自然的一个设计就是，首先把状态和状态对应的行为从原来的大杂烩代码中分离出来，把每个状态所对应的功能处理封装在一个独立的类里面，这样选择不同处理的时候，其实就是在选择不同的状态处理类。

为了统一操作这些不同的状态类，定义一个状态接口来约束它们，这样外部就可以面向这个统一的状态接口编程，而无须关心具体的状态类实现了。

这样一来，要修改某种投票情况所对应的具体功能处理，只需直接修改或者扩展某个状态处理类的功能就可以了。而要添加新的功能就更简单，直接添加新的状态处理类就可以了，当然在使用 Context 的时候，需要设置使用这个新的状态类的实例。

18.2.2 状态模式的结构和说明

状态模式的结构如图 18.1 所示：

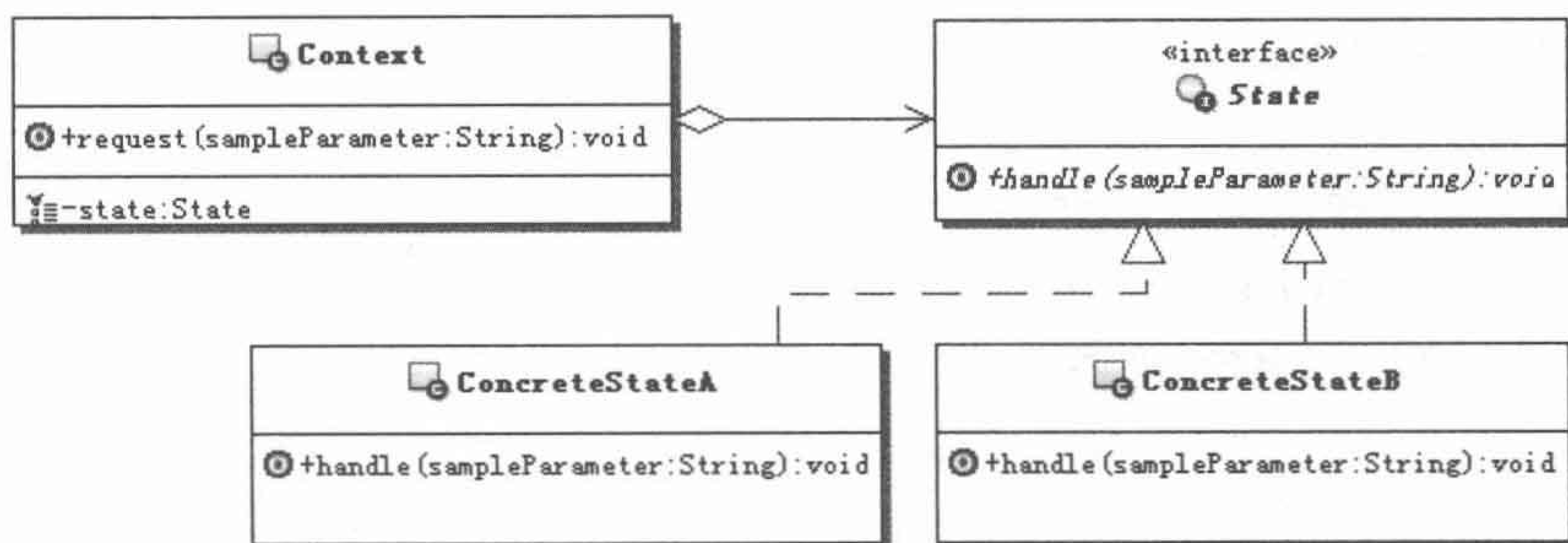


图 18.1　状态模式的结构示意图

- Context：环境，也称上下文，通常用来定义客户感兴趣的接口，同时维护一个来具体处理当前状态的实例对象。
- State：状态接口，用来封装与上下文的一个特定状态所对应的行为。
- ConcreteState：具体实现状态处理的类，每个类实现一个跟上下文相关的状态的具体处理。

18.2.3 状态模式示例代码

（1）首先来看看状态接口。示例代码如下：

```
/**
 * 封装与Context的一个特定状态相关的行为
 */
public interface State {
    /**
     * 状态对应的处理
     * @param sampleParameter 示例参数，说明可以传入参数，具体传入
     *             什么样的参数，传入几个参数，由具体应用来具体分析
     */
    public void handle(String sampleParameter);
}
```

（2）再来看看具体的状态实现。目前具体的实现 ConcreteStateA 和 ConcreteStateB 示范的是一样的，只是名称不同。示例代码如下：

```
/**
 * 实现一个与Context的一个特定状态相关的行为
 */
public class ConcreteStateA implements State {
    public void handle(String sampleParameter) {
        //实现具体的处理
    }
}
/**
 * 实现一个与Context的一个特定状态相关的行为
 */
public class ConcreteStateB implements State {
    public void handle(String sampleParameter) {
        //实现具体的处理
    }
}
```

（3）接下来看看上下文的具体实现。上下文通常用来定义客户感兴趣的接口，同时维护一个具体的处理当前状态的实例对象。示例代码如下：

```
/**
 * 定义客户感兴趣的接口，通常会维护一个State类型的对象实例
 */
public class Context {
    /**
     * 持有一个State类型的对象实例
     */
    private State state;
```

```
    /**
     * 设置实现 State 的对象的实例
     * @param state 实现 State 的对象的实例
     */
    public void setState(State state) {
        this.state = state;
    }
    /**
     * 用户感兴趣的接口方法
     * @param sampleParameter 示意参数
     */
    public void request(String sampleParameter) {
        //在处理中，会转调 state 来处理
        state.handle(sampleParameter);
    }
}
```

18.2.4　使用状态模式重写示例

看完了上面的状态模式的知识，有些朋友跃跃欲试，打算使用状态模式来重写前面的示例。要想使用状态模式，首先需要把投票过程的各种状态定义出来，然后把这些状态对应的处理从原来大杂烩的实现中分离出来，形成独立的状态处理对象。而原来投票管理的对象就相当于 Context 了。

把状态对应的行为分离出去以后，怎么调用呢？

按照状态模式的示例，是在 Context 中处理客户请求的时候，转调相应的状态对应的具体的状态处理类来进行处理。

那就引出下一个问题：那么这些状态怎么变化呢？

看原来的实现，就是在投票方法中，根据投票的次数进行判断，并维护投票类型的变化。那好，也依葫芦画瓢，就在投票方法中来维护状态变化。

这个时候的程序结构如图 18.2 所示。

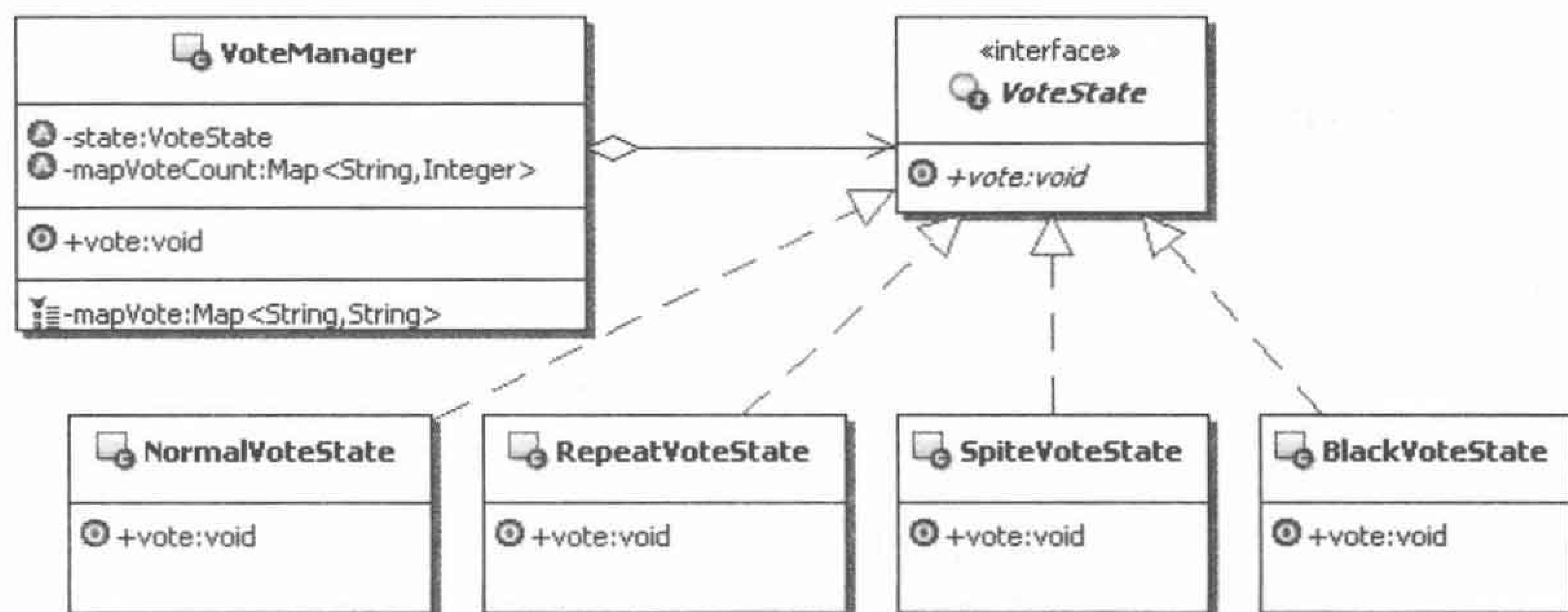


图 18.2　状态模式的示例程序机构示意图

（1）先来看看状态接口的代码实现。示例代码如下：

```
/**
 * 封装一个投票状态相关的行为
 */
public interface VoteState {
    /**
     * 处理状态对应的行为
     * @param user 投票人
     * @param voteItem 投票项
     * @param voteManager 投票上下文，用来在实现状态对应的功能处理的时候，
     *                    可以回调上下文的数据
     */
    public void vote(String user,String voteItem
                            ,VoteManager voteManager);
}
```

（2）定义了状态接口，那就该来看看如何实现各个状态对应的处理了。现在的实现很简单，就是把原来的实现从投票管理类中分离出来就可以了。

下面来看看正常投票状态对应的处理。示例代码如下：

```
public class NormalVoteState implements VoteState{
    public void vote(String user, String voteItem
                            ,VoteManager voteManager) {
        //正常投票
        //记录到投票记录中
        voteManager.getMapVote().put(user, voteItem);
        System.out.println("恭喜你投票成功");
    }
}
```

下面来看看重复投票状态对应的处理。示例代码如下：

```
public class RepeatVoteState implements VoteState{
    public void vote(String user, String voteItem
,VoteManager voteManager) {
        //重复投票
        //暂时不做处理
        System.out.println("请不要重复投票");
    }
}
```

下面来看看恶意投票状态对应的处理。示例代码如下：

```
public class SpiteVoteState implements VoteState{
```

```
    public void vote(String user, String voteItem
                              ,VoteManager voteManager) {
        //恶意投票
        //取消用户的投票资格，并取消投票记录
        String s = voteManager.getMapVote().get(user);
        if(s!=null){
            voteManager.getMapVote().remove(user);
        }
        System.out.println("你有恶意刷票行为，取消投票资格");
    }
}
```

下面来看看黑名单状态对应的处理。示例代码如下：

```
public class BlackVoteState implements VoteState{
    public void vote(String user, String voteItem
                              ,VoteManager voteManager) {
        //黑名单
        //记入黑名单中，禁止登录系统了
        System.out.println("进入黑名单，将禁止登录和使用本系统");
    }
}
```

（3）定义好了状态接口并实现了各个状态对应的处理，看看现在的投票管理，相当于状态模式中的上下文，相对而言，它的改变如下。

- 添加了持有状态处理对象。
- 添加了能获取记录用户投票结果的 Map 的方法，各个状态处理对象，在进行状态对应处理的时候，需要获取上下文中的记录用户投票结果的 Map 数据。
- 在 vote()方法实现中，原来判断投票的类型就变成了判断投票的状态；而原来每种投票类型对应的处理，现在已经封装到对应的状态对象中去了，因此直接转调对应的状态对象的方法即可。

示例代码如下：

```
/**
 * 投票管理
 */
public class VoteManager {
    /**
     * 持有状态处理对象
     */
    private VoteState state = null;
    /**
     * 记录用户投票的结果，Map<String,String>对应 Map<用户名称,投票的选项>
```

```
 */
private Map<String,String> mapVote =
                              new HashMap<String,String>();
/**
 * 记录用户投票次数，Map<String,Integer>对应Map<用户名称,投票的次数>
 */
private Map<String,Integer> mapVoteCount =
                              new HashMap<String,Integer>();
/**
 * 获取记录用户投票结果的Map
 * @return 记录用户投票结果的Map
 */
public Map<String, String> getMapVote() {
    return mapVote;
}

/**
 * 投票
 * @param user 投票人，为了简单，就是用户名称
 * @param voteItem 投票的选项
 */
public void vote(String user,String voteItem){
    //1：先为该用户增加投票的次数
    //从记录中取出已有的投票次数
    Integer oldVoteCount = mapVoteCount.get(user);
    if(oldVoteCount==null){
        oldVoteCount = 0;
    }
    oldVoteCount = oldVoteCount + 1;
    mapVoteCount.put(user, oldVoteCount);
    //2：判断该用户投票的类型，就相当于是判断对应的状态
    //到底是正常投票、重复投票、恶意投票还是上黑名单的状态
    if(oldVoteCount==1){
        state = new NormalVoteState();
    }else if(oldVoteCount>1 && oldVoteCount<5){
        state = new RepeatVoteState();
    }else if(oldVoteCount >= 5 && oldVoteCount<8){
        state = new SpiteVoteState();
    }else if(oldVoteCount>=8){
        state = new BlackVoteState();
```

```
        }

        //然后转调状态对象来进行相应的操作
        state.vote(user, voteItem, this);
    }
}
```

（4）该写个客户端来测试一下了，经过这样修改，好用吗？试试看就知道了。客户端没有任何的改变，跟前面实现的一样。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        VoteManager vm = new VoteManager();
        for(int i=0;i<8;i++){
            vm.vote("u1", "A");
        }
    }
}
```

运行一下试试吧，结果应该是跟前面一样的，也就是说都是实现一样的功能，只是采用了状态模式来实现。测试结果如下：

```
恭喜你投票成功
请不要重复投票
请不要重复投票
请不要重复投票
你有恶意刷票行为，取消投票资格
你有恶意刷票行为，取消投票资格
你有恶意刷票行为，取消投票资格
进入黑名单，将禁止登录和使用本系统
```

从上面的示例可以看出，状态的转换基本上都是内部行为，主要在状态模式内部来维护。比如对于投票的人员，任何时候他的操作都是投票，但是投票管理对象的处理却不一定一样，会根据投票的次数来判断状态，然后根据状态去选择不同的处理。

18.3　模 式 讲 解

18.3.1　认识状态模式

1. 状态和行为

所谓对象的状态，通常指的就是对象实例的属性的值；而行为指的就是对象的功能，再具体点说，行为大多可以对应到方法上。

状态模式的功能就是分离状态的行为，通过维护状态的变化，来调用不同状态对应

的不同功能。

也就是说，状态和行为是相关联的，它们的关系可以描述为：**状态决定行为**。

由于状态是在运行期被改变的，因此行为也会在运行期根据状态的改变而改变，看起来，同一个对象，在不同的运行时刻，行为是不一样的，就像是类被修改了一样。

2. 行为的平行性

注意是平行性而不是平等性。所谓平行性指的是各个状态的行为所处的层次是一样的，相互是独立的、没有关联的，是根据不同的状态来决定到底走平行线的哪一条。行为是不同的，当然对应的实现也是不同的，相互之间是不可替换的，如图 18.3 所示。

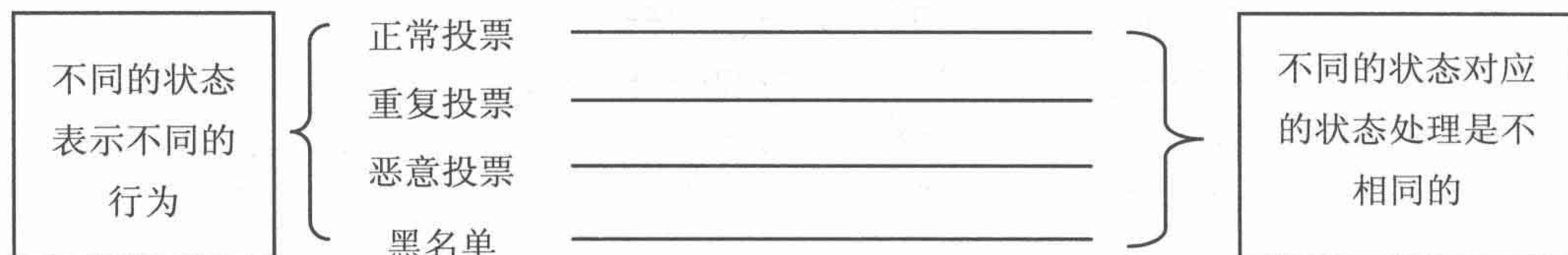


图 18.3　状态的平行性示意图

而平等性强调的是可替换性，大家是同一行为的不同描述或实现，因此在同一个行为发生的时候，可以根据条件挑选任意一个实现来进行相应的处理，如图 18.4 所示。

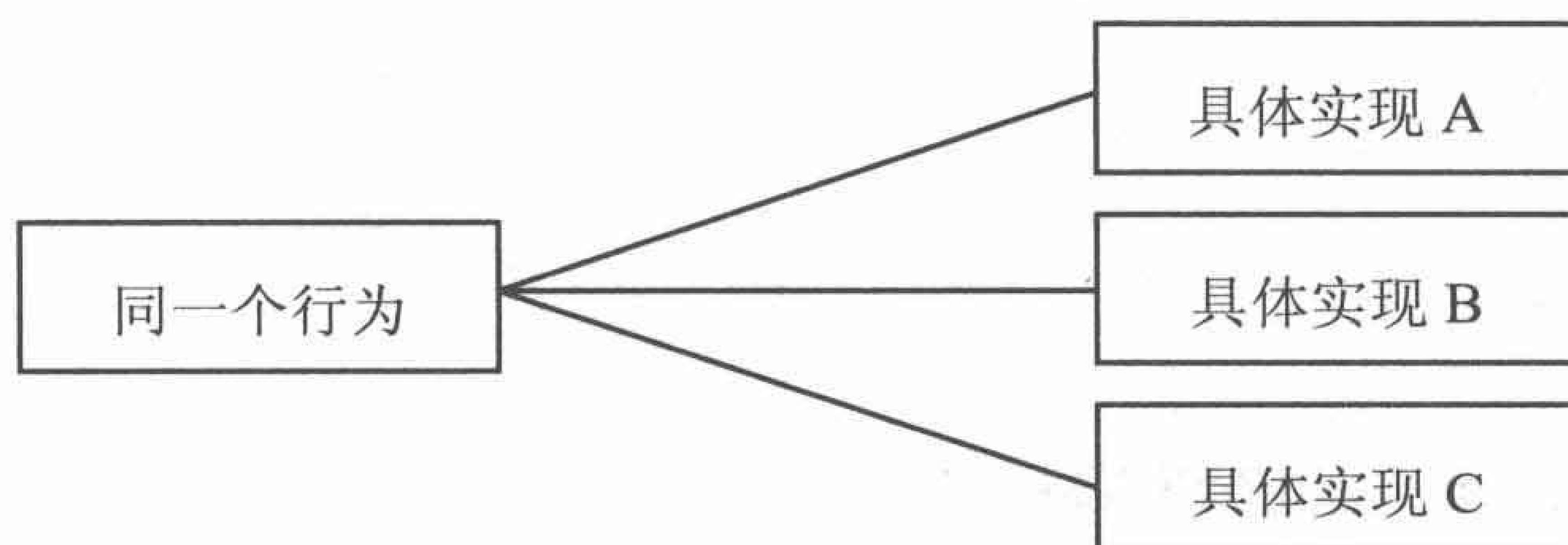


图 18.4　平等性的示意图

大家可能会发现状态模式的结构和策略模式的结构完全一样，但是，它们的目的、实现、本质却是完全不一样的。还有行为之间的特性也是状态模式和策略模式一个很重要的区别，状态模式的行为是平行性的，不可相互替换的；而策略模式的行为是平等性的，是可以相互替换的。

3. 上下文和状态处理对象

在状态模式中，上下文是持有状态的对象，但是上下文自身并不处理跟状态相关的行为，而是把处理状态的功能委托给了状态对应的状态处理类来处理。

在具体的状态处理类中经常需要获取上下文自身的数据，甚至在必要的时候会回调上下文的方法，因此，通常将上下文自身当作一个参数传递给具体的状态处理类。

客户端一般只和上下文交互。客户端可以用状态对象来配置一个上下文，一旦配置完毕，就不再需要和状态对象打交道了。客户端通常不负责运行期间状态的维护，也不负责决定后续到底使用哪一个具体的状态处理对象。

4. 不完美的 OCP 体验

好了，已经使用状态模式重写了前面的示例，那么到底能不能解决前面提出的问题呢？也就是修改和扩展是否方便？一起来看看。

先修改已有的功能吧。由于现在每个状态对应的处理已经封装到对应的状态类中了，要修改已有的某个状态的功能，直接扩展某个类进行修改就可以了，对其他的程序没有影响。比如，现在要修改正常投票状态对应的功能，对正常投票的用户给予积分奖励，那么只需要扩展正常投票状态对应的类，然后进行修改即可。示例代码如下：

```
public class NormalVoteState2 extends NormalVoteState{
    public void vote(String user, String voteItem
                                        , VoteManager voteManager) {
        //先调用已有的功能
        super.vote(user, voteItem, voteManager);
        //给予积分奖励，示意一下
        System.out.println("奖励积分10分");
    }
}
```

一切良好，对吧，可是怎么让 VoteManager 使用这个新的实现类呢？按照目前的实现，没有办法，只好去修改 VoteManager 的 vote()方法中对状态的维护代码了，把使用 NormalVoteState 的地方换成使用 NormalVoteState2。

再看看如何添加新的功能，比如投票超过 8 次但不足 10 次的，给个机会，只是禁止登录和使用系统 3 天，如果再犯，才进入黑名单。要实现这个功能，先要对原来的投票超过 8 次进入黑名单的功能进行修改，修改成投票超过 10 次才进入黑名单；然后新加入一个功能，实现超过 8 次但不足 10 次的，只是禁止登录和使用系统 3 天的功能。把这个新功能实现出来。示例代码如下：

```
public class BlackWarnVoteState implements VoteState{
    public void vote(String user, String voteItem
                                        , VoteManager voteManager) {
        //待进黑名单警告状态
        System.out.println("禁止登录和使用系统3天");
    }
}
```

实现好了这个类，该怎样加入到已有的系统呢？

同样需要去修改上下文的 vote()方法中对于状态判断和维护的代码，示例代码如下：

```
if(oldVoteCount==1){
        state = new NormalVoteState2();
}else if(oldVoteCount>1 && oldVoteCount<5){
        state = new RepeatVoteState();
}else if(oldVoteCount >= 5 && oldVoteCount<8){
        state = new SpiteVoteState();
}else if(oldVoteCount>=8 && oldVoteCount<10){
        state = new BlackWarnVoteState();
```

```
}else if(oldVoteCount>10){
    state = new BlackVoteState();
}
```

好像也实现了功能，而且改动起来确实也变得简单点了，但是仔细想想，是不是没有完全遵循 OCP 原则？结论是很显然的，**明显没有完全遵循 OCP 原则**。

> 延伸 **这里要说明一点，设计原则是大家在设计和开发中尽量去遵守的，但不是一定要遵守，尤其是完全遵守。在实际开发中，完全遵守那些设计原则几乎是不可能完成的任务。**

就像状态模式的实际实现中，由于状态的维护和转换在状态模式结构里面，不管你是扩展了状态实现类，还是新添加了状态实现类，都需要修改状态维护和转换的地方，以使用新的实现。

虽然可以有好几个地方来维护状态的变化（这个后面会讲到），但都是在状态模式结构里面的，所以都有这个问题，算是不完美的 OCP 体验吧。

5. 创建和销毁状态对象

在应用状态模式的时候，有一个常见的考虑，那就是：究竟何时创建和销毁状态对象？常见的有以下几个选择。

- 当需要使用状态对象的时候创建，使用完后就销毁它们。
- 提前创建它们并且始终不销毁。
- 采用延迟加载和缓存合用的方式，就是当第一次需要使用状态对象的时候创建，使用完后并不销毁对象，而是把这个对象缓存起来，等待下一次使用，而且在合适的时候，会由缓存框架销毁状态对象。

怎么选择呢？下面给出选择建议。

如果要进入的状态在运行时是不可知的，而且上下文是比较稳定的，不会经常改变状态，而且使用也不频繁，这个时候建议选择第一种方案。

如果状态改变很频繁，也就是需要频繁地创建状态对象，而且状态对象还存储着大量的数据信息，这种情况建议选择第二种方案。

如果无法确定状态改变是否频繁，而且有些状态对象的状态数据量大，有些比较小，一切都是未知的，建议选择第三种方案。

事实上，在实际工程开发过程中，第三种方案是首选。因为它兼顾了前面两种方案的优点，而又避免了它们的缺点，几乎能适应各种情况的需要。只是这个方案在实现的时候，需要实现一个合理的缓存框架，而且要考虑多线程并发的问题，因为需要由缓存框架来在合适的时候销毁状态对象，因此实现上难度稍大。另外在实现中还可以考虑结合享元模式，通过享元模式来共享状态对象。

6. 状态模式的调用顺序示意图

状态模式在实现上，对于状态的维护有不同的实现方式。前面的示例中，采用的是

在 Context 中进行状态的维护和转换。这里先画出这种方式的调用顺序示意图，其他的方式则在后面讲到时再画。

在 Context 中进行状态维护和转换的调用顺序如图 18.5 所示。

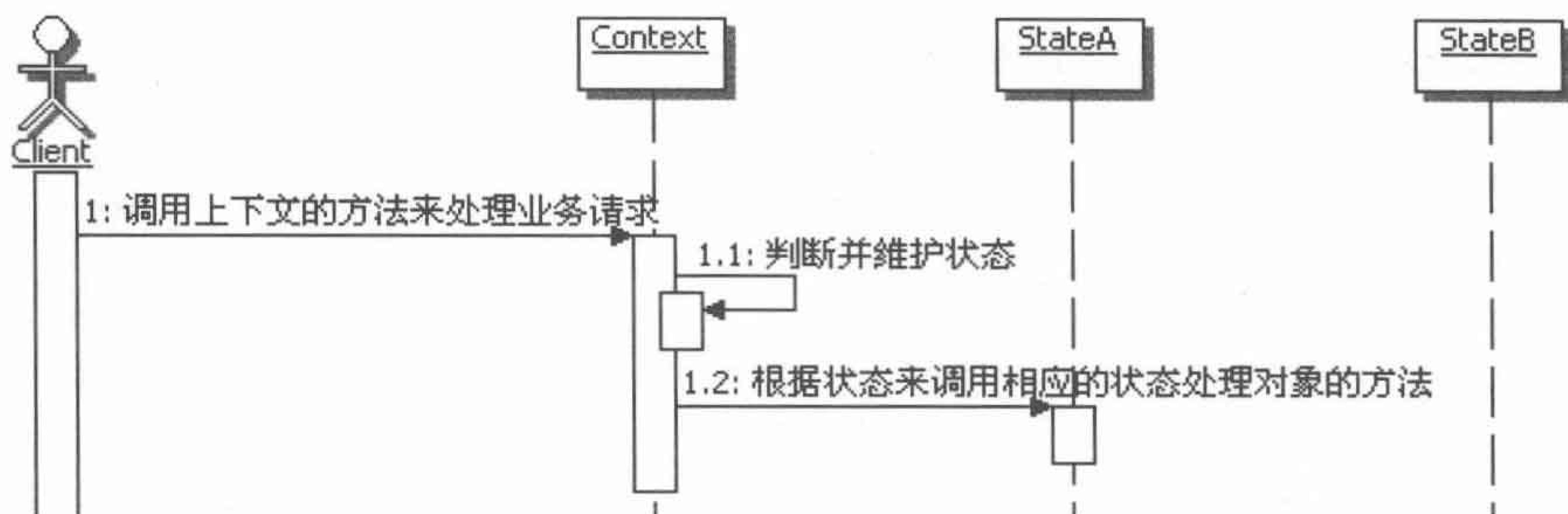


图 18.5　在 Context 中进行状态维护和转换的调用顺序示意图

18.3.2　状态的维护和转换控制

所谓状态的维护，指的是维护状态的数据，给状态设置不同的状态值；而状态的转换，指的是根据状态的变化来选择不同的状态处理对象。在状态模式中，通常有两个地方可以进行状态的维护和转换控制。

一个就是在上下文中。因为状态本身通常被实现为上下文对象的状态，因此可以在上下文中进行状态维护，当然也就可以控制状态的转换了。前面投票的示例就是采用这种方式。

另外一个地方就是在状态的处理类中。当每个状态处理对象处理完自身状态所对应的功能后，可以根据需要指定后继状态，以便让应用能正确处理后续的请求。

先看看示例。为了对比学习，下面来看看如何把前面投票的例子修改成：在状态处理类中进行后续状态的维护和转换。

（1）同样先来看投票状态的接口。没有什么变化。示例代码如下：

```
/**
 * 封装一个投票状态相关的行为
 */
public interface VoteState {
    /**
     * 处理状态对应的行为
     * @param user 投票人
     * @param voteItem 投票项
     * @param voteManager 投票上下文，用来在实现状态对应的功能处理的时候，
     *                    可以回调上下文的数据
     */
    public void vote(String user,String voteItem
                                        ,VoteManager voteManager);
}
```

（2）对于各个具体的状态实现对象，主要的变化在于：在处理完自己状态对应的功能后，还需要维护和转换状态对象。

一个一个来看吧，先看看正常投票的状态处理对象。示例代码如下：

```
public class NormalVoteState implements VoteState{
    public void vote(String user, String voteItem
                                    , VoteManager voteManager) {
        //正常投票，记录到投票记录中
        voteManager.getMapVote().put(user, voteItem);
        System.out.println("恭喜你投票成功");
        //正常投票完成，维护下一个状态，同一个人再投票就重复了
        voteManager.getMapState().put(user,new RepeatVoteState());
    }
}
```

再来看看重复投票状态对应的处理对象。示例代码如下：

```
public class RepeatVoteState implements VoteState{
    public void vote(String user, String voteItem
                                    , VoteManager voteManager) {
        //重复投票，暂时不做处理
        System.out.println("请不要重复投票");
        //重复投票完成，维护下一个状态，重复投票到5次，就算恶意投票了
        //注意这里是判断大于等于4，因为这里设置的是下一个状态
        //下一个操作次数就是5了，就应该算是恶意投票了
        if(voteManager.getMapVoteCount().get(user) >= 4){
            voteManager.getMapState().put(user,
                                    new SpiteVoteState());
        }
    }
}
```

接下来看看恶意投票状态对应的处理对象。示例代码如下：

```
public class SpiteVoteState implements VoteState{
    public void vote(String user, String voteItem
                                    , VoteManager voteManager) {
        //恶意投票，取消用户的投票资格，并取消投票记录
        String s = voteManager.getMapVote().get(user);
        if(s!=null){
            voteManager.getMapVote().remove(user);
        }
        System.out.println("你有恶意刷票行为，取消投票资格");
        //恶意投票完成，维护下一个状态，投票到8次，就进黑名单了
```

```
        //注意这里是判断大于等于7，因为这里设置的是下一个状态
        //下一个操作次数就是8了，就应该算是进黑名单了
        if(voteManager.getMapVoteCount().get(user) >= 7){
            voteManager.getMapState().put(user,
                                                    new BlackVoteState());
        }
    }
}
```

下面来看看黑名单状态对应的处理对象。没什么变化。示例代码如下：

```
public class BlackVoteState implements VoteState{
    public void vote(String user, String voteItem
                                        , VoteManager voteManager) {
        //黑名单，记入黑名单中，禁止登录系统了
        System.out.println("进入黑名单，将禁止登录和使用本系统");
    }
}
```

（3）下面来看看现在的投票管理类该如何实现了？和在上下文中维护和转换状态相比，大致有如下的变化。

- 需要按照每个用户来记录他们对应的投票状态，不同的用户，对应的投票状态是不同的，因此使用一个 Map 来记录，而不再是原来的一个单一的投票状态对象。

> 提示　**可能有些朋友会问，那为什么前面的实现可以呢？那是因为投票状态是由投票管理对象集中控制的，不同的人员在进入投票方法的时候，是重新判断该人员具体的状态对象的，而现在是要把状态维护分散到各个状态类中，因此需要记录各个状态类判断以后的结果。**

- 需要把记录投票状态的数据，还有记录投票次数的数据，提供相应的 getter 方法，各个状态在处理的时候需要通过这些方法来访问数据。
- 将原来在 vote()方法中进行的状态控制和转换删除，变成直接根据人员来从状态记录的 Map 中获取对应的状态对象了。

看看实现代码吧。示例代码如下：

```
public class VoteManager {
    /**
     * 记录当前每个用户对应的状态处理对象，每个用户当前的状态是不同的
     * Map<String,VoteState>对应 Map<用户名称,当前对应的状态处理对象>
     */
    private Map<String,VoteState> mapState =
                                    new HashMap<String,VoteState>();
    /**
```

```
     * 记录用户投票的结果，Map<String,String>对应Map<用户名称,投票的选项>
     */
    private Map<String,String> mapVote =
                                new HashMap<String,String>();
    /**
     * 记录用户投票次数，Map<String,Integer>对应Map<用户名称,投票的次数>
     */
    private Map<String,Integer> mapVoteCount =
                                new HashMap<String,Integer>();
    /**
     * 获取记录用户投票结果的Map
     * @return 记录用户投票结果的Map
     */
    public Map<String, String> getMapVote() {
        return mapVote;
    }
    /**
     * 获取记录每个用户对应的状态处理对象的Map
     * @return 记录每个用户对应的状态处理对象的Map
     */
    public Map<String, VoteState> getMapState() {
        return mapState;
    }
    /**
     * 获取记录每个用户对应的投票次数的Map
     * @return 记录每个用户对应的投票次数的Map
     */
    public Map<String, Integer> getMapVoteCount() {
        return mapVoteCount;
    }
    /**
     * 投票
     * @param user 投票人，为了简单，就是用户名称
     * @param voteItem 投票的选项
     */
    public void vote(String user,String voteItem){
        //1：先为该用户增加投票的次数
        //从记录中取出已有的投票次数
        Integer oldVoteCount = mapVoteCount.get(user);
        if(oldVoteCount==null){
```

```
            oldVoteCount = 0;
        }
        oldVoteCount = oldVoteCount + 1;
        mapVoteCount.put(user, oldVoteCount);

        //2：获取该用户的投票状态
        VoteState state = mapState.get(user);
        //如果没有投票状态，说明还没有投过票，就初始化一个正常投票状态
        if(state==null){
            state = new NormalVoteState();
        }

        //然后转调状态对象来进行相应的操作
        state.vote(user, voteItem, this);
    }
}
```

（4）实现的差不多了，该来测试了，客户端没有变化，去运行一下，看看效果。看看两种维护状态变化的方式实现的结果一样吗？答案应该是一样的。

那么到底如何选择这两种方式呢？

- 如果状态转换的规则是一定的，一般不需要进行什么扩展规则，那么就适合在上下文中统一进行状态的维护。
- 如果状态的转换取决于前一个状态动态处理的结果，或者是依赖于外部数据，为了增强灵活性，这种情况下，一般是在状态处理类中进行状态的维护。

（5）采用让状态对象来维护和转换状态的调用顺序如图 18.6 所示。

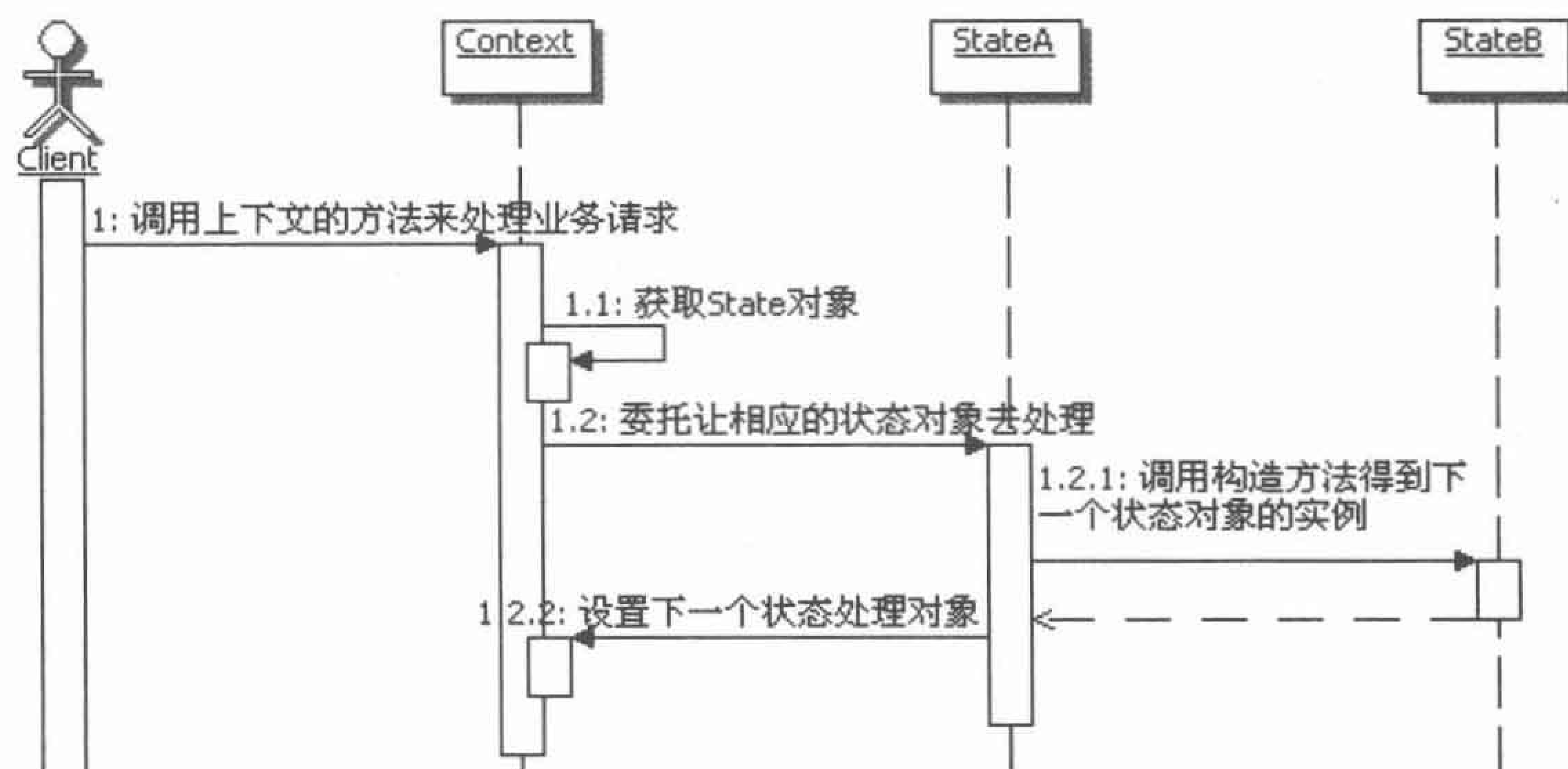


图 18.6　状态对象来维护和转换状态的调用顺序示意图

（6）再来看看这种实现方式下，如何修改已有的功能，或者是添加新的状态处理。

要修改已有的功能，同样是找到对应的状态处理对象，要么直接修改，要么扩展。前面已经示例过了，这里不再赘述。

对于添加新的状态处理的功能，这种实现方式会比较简单。先直接添加新的状态处

理的类，然后找到需要转换到这个新状态的状态处理类，修改那个处理类，让其转换到这个新状态就可以了。

比如还是来实现：投票超过 8 次但不足 10 次的，给个机会，只是禁止登录和使用系统 3 天，如果再犯，才进入黑名单的功能。按照现在的方式，示例代码如下：

```
public class BlackWarnVoteState implements VoteState{
    public void vote(String user, String voteItem
                                , VoteManager voteManager) {
        //待进黑名单警告状态
        System.out.println("禁止登录和使用系统3天");
        //待进黑名单警告处理完成，维护下一个状态，投票到 10 次，就进黑名单了
        //注意这里是判断大于等于 9，因为这里设置的是下一个状态
        //下一个操作次数就是 10 了，就应该算是进黑名单了
        if(voteManager.getMapVoteCount().get(user) >= 9){
            voteManager.getMapState().put(user
                            , new BlackVoteState());
        }
    }
}
```

那么如何加入系统呢？

不再是去修改 VoteManger 了，而是找到需要转换到这个新状态的那个状态，修改它的状态维护和转换。应该是在恶意投票处理中，让它转换到这个新的状态，也就是把恶意投票处理中的下面这句话：

```
voteManager.getMapState().put(user, new BlackVoteState());
```

替换成：

```
voteManager.getMapState().put(user, new BlackWarnVoteState());
```

这样就自然地把现在新的状态处理添加到了已有的应用中。

18.3.3 使用数据库来维护状态

在实际开发中，还有一个方式来维护状态，那就是使用数据库，在数据库中存储下一个状态的识别数据。也就是说，维护下一个状态演化成了维护下一个状态的识别数据，比如状态编码。

这样，在程序中通过查询数据库中的数据来得到状态编码，然后根据状态编码来创建出相应的状态对象，再委托相应的状态对象进行功能处理。

还是用前面投票的示例来说明，如果使用数据库来维护状态的话，大致如何实现。

（1）首先，就是每个具体的状态处理类中，原本在处理完成后，要判断下一个状态是什么，然后创建下一个状态对象，并设置回到上下文中；但是如果使用数据库的方式，那就不用创建下一个状态对象，也不用设置回到上下文中了，而是把下一个状态对应的

编码记入数据库中，这样就可以了。

还是示意一个，看看重复投票状态下的实现吧。示例代码如下：

```
public class RepeatVoteState implements VoteState{
    public void vote(String user, String voteItem
                                        , VoteManager voteManager) {
        //重复投票，暂时不做处理
        System.out.println("请不要重复投票");
        //重复投票完成，维护下一个状态，重复投票到 5 次，就算恶意投票了
        if(voteManager.getMapVoteCount().get(user) >= 4){
            voteManager.getMapState().put(user,
                                        new SpiteVoteState());
            //直接把下一个状态的编码记录到数据库就可以了
        }
    }
}
```

这里只是示意一下，并不真的去写和数据库操作相关的代码。其他的状态实现类，也做同样类似的修改，这里不再赘述。

（2）在 Context 中，也就是投票管理对象中，则不需要那个记录所有用户状态的 Map 了，直接从数据库中获取该用户当前对应的状态编码，然后根据状态编码来创建出状态对象来。原有的示例代码如下：

```
//2：获取该用户的投票状态
VoteState state = mapState.get(user);
//如果没有投票状态，说明还没有投过票，就初始化一个正常投票状态
if(state==null){
state = new NormalVoteState();
}
```

现在被修改了。示例代码如下：

```
VoteState state = null;
//2：直接从数据库获取该用户对应的下一个状态的状态编码
String stateId = "从数据库中获取这个状态编码";
//开始根据状态编码来创建需用的状态对象
if(stateId==null || stateId.trim().length()==0){
//如果没有值，说明还没有投过票，就初始化一个正常投票状态
     state = new NormalVoteState();
}else if("重复投票".equals(stateId)){
     state = new RepeatVoteState();
}else if("恶意投票".equals(stateId)){
     state = new SpiteVoteState();
}else if("黑名单".equals(stateId)){
```

```
        state = new BlackVoteState();
}
```

可能有些朋友会发现，如果向数据库中存储下一个状态对象的状态编码，那么上下文中就不需要再持有状态对象了，相当于把这个功能放到数据库中了。有那么点相似性，不过要注意，数据库存储的只是状态编码，而不是状态对象，获取到数据库中的状态编码后，在程序中仍然需要根据状态编码去真正创建对应的状态对象。

当然，要想程序更通用一点，可以通过配置文件来配置状态编码和对应的状态处理类，也可以直接在数据库中记录状态编码和对应的状态处理类。这样的话，在上下文中，先获取下一个状态的状态编码，然后根据这个状态编码去获取对应的类，再通过反射来创建对象，如此实现就避免了那一长串的 if-else，而且以后添加新的状态编码和状态处理对象也不用再修改代码了。示例代码如下：

```
VoteState state = null;
//2：直接从数据库获取该用户对应的下一个状态的状态编码
String stateId = "从数据库中获取这个值";
//开始根据状态编码来创建需用的状态对象

//根据状态编码去获取相应的类
String className = "根据状态编码去获取相应的类";
//使用反射创建对象实例，简单示意一下
Class c = Class.forName(className);
state = (VoteState)c.newInstance();
```

（3）直接把“转移”记录到数据库中。

还有一种情况是直接把“转移”记录到数据库中，这样会更灵活。所谓转移，指的就是描述从 A 状态到 B 状态的转换变化。

比如，在正常投票状态处理对象中指定使用“转移 A”，而“转移 A”描述的就是从正常投票状态转换成重复投票状态。这样一来，假如今后想要让正常投票处理以后变换成恶意投票状态，就不需要修改程序，而是直接修改数据库中的数据，把数据库中“转移 A”的描述数据修改一下，使其描述从正常投票状态转换成恶意投票状态就可以了。

18.3.4 模拟工作流

做企业应用的朋友，大多数都接触过工作流，至少也处理过业务流程。对于工作流，复杂的应用可能会使用工作流中间件，用工作流引擎来负责流程处理，这个会比较复杂。其实工作流引擎的实现也可以应用状态模式，这里不去讨论。

简单点的，把流程数据存放在数据库中，然后在程序中自己来进行流程控制。对于简单的业务流程控制，可以使用状态模式来辅助进行流程控制，因为大部分这种流程都是状态驱动的。

举个例子来说明吧。比如最常见的“请假流程”，流程是这样的：当某人提出请假申请，先由项目经理审批，如果项目经理不同意，审批就直接结束；如果项目经理同意

了，再看请假的天数是否超过 3 天，项目经理的审批权限只有 3 天以内，如果请假天数在 3 天以内，那么审批也直接结束，否则就提交给部门经理；部门经理审核过后，无论是否同意，审批都直接结束。流程图如图 18.7 所示。

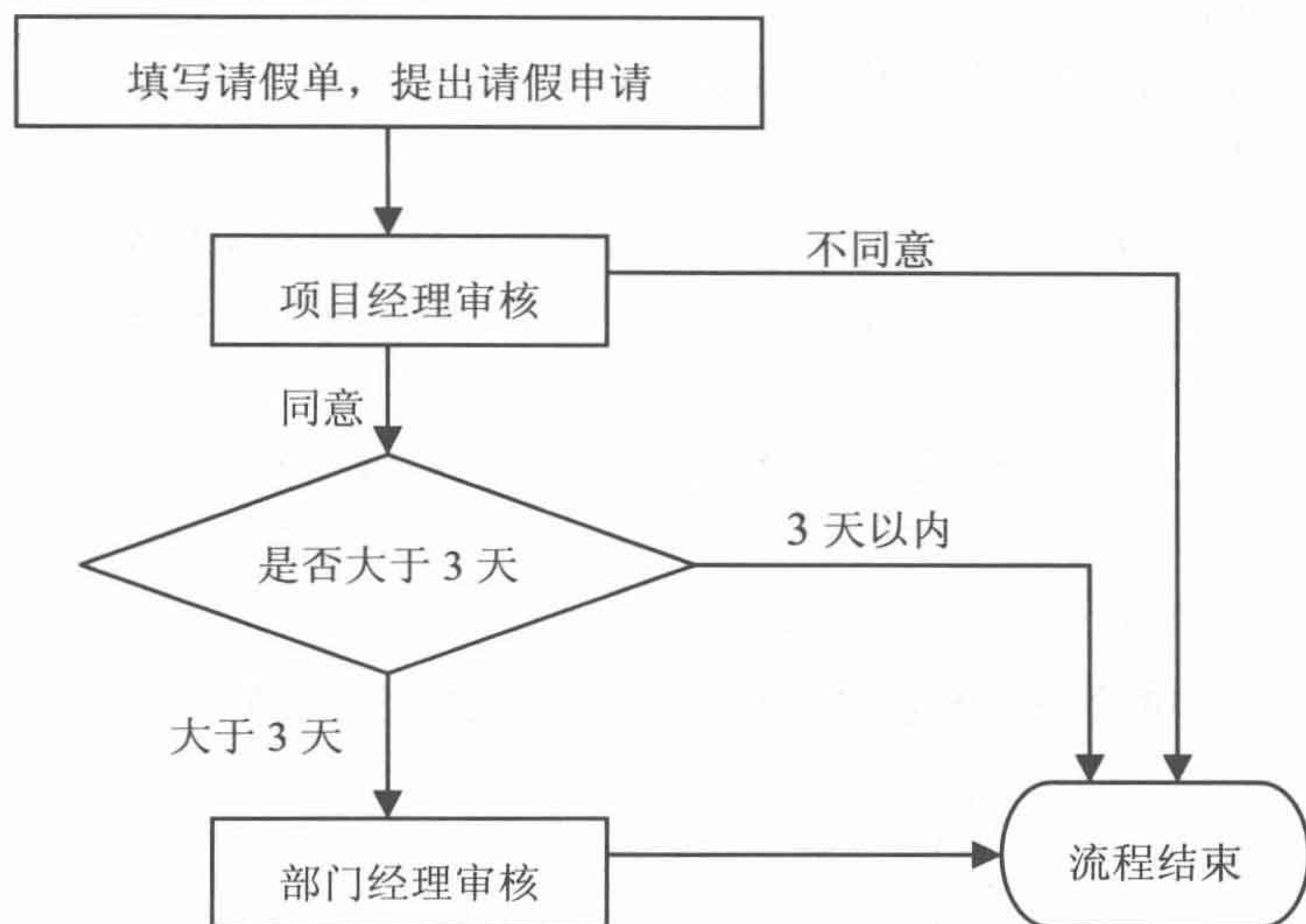


图 18.7　简单的请假流程示意图

在实际开发中，如果不考虑使用工作流软件，按照流程来自己实现的话，这个流程基本的运行过程简单描述如下。

（1）UI 操作：请假人填写请假单，提出请假申请。

（2）后台处理：保存请假单数据到数据库中，然后为项目经理创建一个工作，把工作信息保存到数据库中。

（3）UI 操作：项目经理登录系统，获取自己的工作列表。

（4）后台处理：从数据库中获取相应的工作列表。

（5）UI 操作：项目经理完成审核工作，提交保存。

（6）后台处理：处理项目经理审核的业务，保存审核的信息到数据库。同时判断后续的工作，如果是需要人员参与的，就为参与下一个工作的人员创建工作，把工作信息保存到数据库中。

（7）UI 操作：部门经理登录系统，获取自己的工作列表，基本上是重复第 3 步。

（8）后台处理：从数据库中获取相应的工作列表，基本上是重复第 4 步。

（9）UI 操作：部门经理完成审核工作，提交保存，基本上是重复第 5 步。

（10）后台处理：类推，基本上是重复第 6 步。

1. 实现思路

仔细分析上面的流程图和运行过程，把请假单在流程中的各个阶段的状态分析出来，会发现，整个流程完全可以看成是状态驱动的。

在上面的流程中，请假单大致有如下状态：等待项目经理审核、等待部门经理审核、审核结束。如果用状态驱动来描述上述流程：

（1）当请假人填写请假单，提出请假申请后，请假单的状态是等待项目经理审核状态。

（2）当项目经理完成审核工作，提交保存后，如果项目经理不同意，请假单的状态是审核结束状态；如果项目经理同意，请假天数又在3天以内，请假单的状态是审核结束状态；如果项目经理同意，请假天数大于3天，请假单的状态是等待部门经理审核状态。

（3）当部门经理完成审核工作，提交保存后，无论是否同意，请假单的状态都是审核结束状态。

既然可以把流程看成是状态驱动的，那么就可以自然地使用状态模式，每次当相应的工作人员完成工作，请求流程响应的时候，流程处理的对象会根据当前所处的状态，把流程处理委托给相应的状态对象去处理。

还考虑到在一个系统中会有很多流程，虽然不像通用工作流那么复杂的设计，但还是稍稍提炼一下，至少把各个不同的业务流程，在应用状态模式时的公共功能，或者是架子给搭出来，以便复用这些功能。

（1）这个公共的状态处理机首先提供一个公共的状态处理机。

相当于一个公共的状态模式的Context，在其中提供基本的、公共的功能。这样在实现具体的流程的时候，可以简单一些，对于要求不复杂的流程，甚至可以直接使用。示例代码如下：

```
/**
 * 公共状态处理机，相当于状态模式的Context
 * 包含所有流程使用状态模式时的公共功能
 */
public  class StateMachine {
    /**
     * 持有一个状态对象
     */
    private State state = null;
    /**
     * 包含流程处理需要的业务数据对象，不知道具体类型，为了简单，不使用泛型
     * 用Object，反正只是传递到具体的状态对象中
     */
    private Object businessVO = null;
    /**
     * 执行工作，客户端处理流程的接口方法
     * 在客户完成自己的业务工作后调用
     */
    public void doWork(){
        //转调相应的状态对象真正完成功能处理
        this.state.doWork(this);
    }
```

```
    public State getState() {
        return state;
    }
    public void setState(State state) {
        this.state = state;
    }
    public Object getBusinessVO() {
        return businessVO;
    }
    public void setBusinessVO(Object businessVO) {
        this.businessVO = businessVO;
    }
}
```

（2）来提供公共的状态接口。各个状态对象在处理流程的时候，可以使用统一的接口，可是它们需要的业务数据从何而来呢？那就通过上下文传递过来。示例代码如下：

```
/**
 * 公共状态接口
 */
public interface State {
    /**
     * 执行状态对应的功能处理
     * @param ctx 上下文的实例对象
     */
    public void doWork(StateMachine ctx);
}
```

好了，现在架子已经搭出来了，在实现具体的流程的时候，可以分别扩展它们，来加入跟具体流程相关的功能。

2. 使用状态模式来实现流程

（1）定义请假单的业务数据模型。示例代码如下：

```
public class LeaveRequestModel {
    /**
     * 请假人
     */
    private String user;
    /**
     * 请假开始时间
     */
    private String beginDate;
    /**
```

```
     * 请假天数
     */
    private int leaveDays;
    /**
     * 审核结果
     */
    private String result;
        public String getResult() {
        return result;
    }
    public void setResult(String result) {
        this.result = result;
    }
    public String getUser() {
        return user;
    }
    public String getBeginDate() {
        return beginDate;
    }
    public int getLeaveDays() {
        return leaveDays;
    }
    public void setUser(String user) {
        this.user = user;
    }
    public void setBeginDate(String beginDate) {
        this.beginDate = beginDate;
    }
    public void setLeaveDays(int leaveDays) {
        this.leaveDays = leaveDays;
    }
}
```


（2）定义处理客户端请求的上下文。虽然这里并不需要扩展功能，但还是继承一下状态机，表示可以添加自己的处理。示例代码如下：

```
public class LeaveRequestContext extends StateMachine{
    //这里可以扩展跟自己流程相关的处理
}
```

（3）下面来定义处理请假流程的状态接口。虽然这里并不需要扩展功能，但还是继承一下状态，表示可以添加自己的处理。示例代码如下：

```
public interface LeaveRequestState extends State{
    //这里可以扩展跟自己流程相关的处理
}
```

（4）下面该来实现各个状态具体的处理对象了。

先看看处理项目经理审核的状态类的实现。示例代码如下：

```
/**
 * 处理项目经理的审核，处理后可能对应部门经理审核或者审核结束之中的一种
 */
public class ProjectManagerState implements LeaveRequestState{
    public void doWork(StateMachine request) {
        //先把业务对象造型回来
        LeaveRequestModel lrm =
(LeaveRequestModel)request.getBusinessVO();
        //业务处理，把审核结果保存到数据库中

        //根据选择的结果和条件来设置下一步
        if("同意".equals(lrm.getResult())){
            if(lrm.getLeaveDays() > 3){
                //如果请假天数大于 3 天，而且项目经理同意了，就提交给部门经理
                request.setState(new DepManagerState());
            //为部门经理增加一个工作
            }else{
                //3 天以内的请假，由项目经理做主，
               //就不用提交给部门经理了，转向审核结束状态
                request.setState(new  AuditOverState());
                //给申请人增加一个工作，让他查看审核结果
            }
        }else{
            //项目经理不同意的话，也就不用提交给部门经理了，转向审核结束状态
            request.setState(new  AuditOverState());
           //给申请人增加一个工作，让他查看审核结果
        }
    }
}
```

接下来看看处理部门经理审核的状态类的实现，示例代码如下：

```
/**
 * 处理部门经理的审核，处理后对应审核结束状态
 */
public class DepManagerState implements LeaveRequestState{
```

```
    public void doWork(StateMachine request) {
        //先把业务对象造型回来
        LeaveRequestModel lrm =
                    (LeaveRequestModel)request.getBusinessVO();
        //业务处理，把审核结果保存到数据库中

        //部门经理审核以后，直接转向审核结束状态了
        request.setState(new AuditOverState());
        //给申请人增加一个工作，让他查看审核结果
    }
}
```

再来看看处理审核结束的状态类的实现。示例代码如下：

```
/**
 * 处理审核结束的类
 */
public class AuditOverState implements LeaveRequestState{
    public void doWork(StateMachine request) {
        //先把业务对象造型回来
        LeaveRequestModel lrm =
                    (LeaveRequestModel)request.getBusinessVO();
        //业务处理，在数据中记录整个流程结束
    }
}
```

（5）由于上面的实现中，涉及大量需要数据库支持的功能，同时还需要提供页面来让用户操作，才能驱动流程运行，所以无法像其他示例那样，写个客户端就能进行测试。当然这个可以在后面稍稍改变，模拟一下实现，就可以运行来看效果了。

先来看看此时用状态模式实现的这个流程的程序结构示意图，如图 18.8 所示。

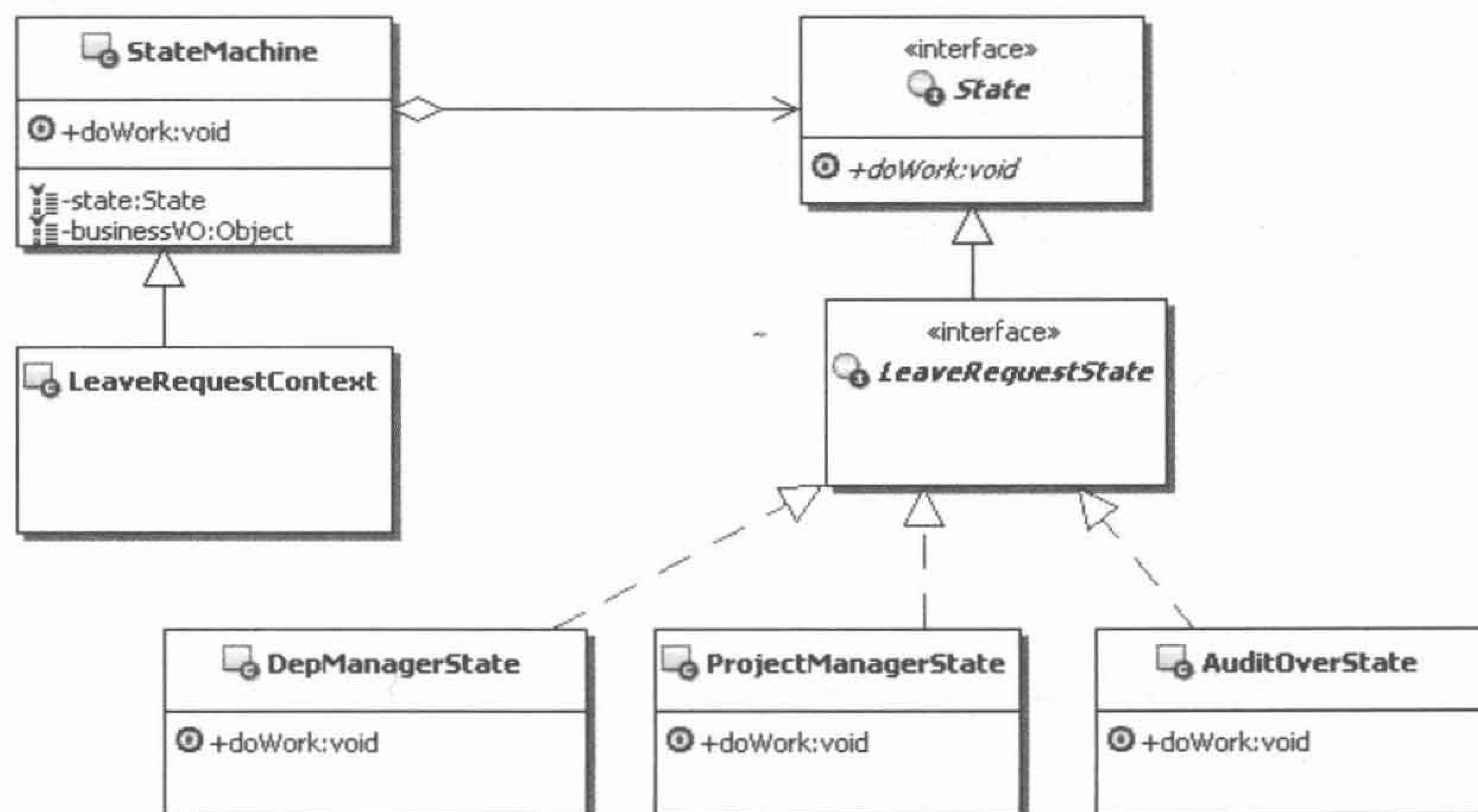


图 18.8　用状态模式实现流程的程序结构示意图

下面来看看怎么改造上面的示例，让它能运转起来。这样更加有利于大家来体会在处理这种流程的应用中，如何使用状态模式。

3. 改进上面使用状态模式实现流程的示例

上面的示例不能运行有两个基本原因：一是没有数据库实现部分，二是没有界面。要解决这个问题，那就采用字符界面，来让客户输入数据；另外把运行放到同一个线程中，这样就不存在传递数据的问题，也就不需要保存数据了，因为数据在内存中。

原来是提交了请假申请，把数据保存在数据库中，然后项目经理从数据库中获取这些数据。现在一步到位，直接把申请数据传递过去，就可以处理了。

（1）根据上面的思路，其实也就是来修改那几个状态处理对象的实现。

先看看处理项目经理审核的状态类的实现，使用 Scanner 接受命令行输入数据。示例代码如下：

```
import java.util.Scanner;
/**
 * 处理项目经理的审核，处理后可能对应部门经理审核或者审核结束之中的一种
 */
public class ProjectManagerState implements LeaveRequestState{
    public void doWork(StateMachine request) {
        //先把业务对象造型回来
        LeaveRequestModel lrm =
                (LeaveRequestModel)request.getBusinessVO();
        System.out.println("项目经理审核中，请稍候......");
        //模拟用户处理界面，通过控制台来读取数据
        System.out.println(lrm.getUser()+"申请从"
            +lrm.getBeginDate()+"开始请假"+lrm.getLeaveDays()
            +"天,请项目经理审核(1为同意，2为不同意)：");
        //读取从控制台输入的数据
        Scanner scanner = new Scanner(System.in);
        if(scanner.hasNext()){
            int a = scanner.nextInt();
            //设置回到上下文中
            String result = "不同意";
            if(a==1){
                result = "同意";
            }
            lrm.setResult("项目经理审核结果："+result);
            //根据选择的结果和条件来设置下一步
            if(a==1){
                if(lrm.getLeaveDays() > 3){
                    //如果请假天数大于 3 天，而且项目经理同意了，
```

```
                    //就提交给部门经理
                     request.setState(new DepManagerState());
                     //继续执行下一步工作
                     request.doWork();
                 }else{
                     //3天以内的请假，由项目经理做主，就不用提交给部门经理了，
                    //转向审核结束状态
                     request.setState(new  AuditOverState());
                     //继续执行下一步工作
                     request.doWork();
                 }
             }else{
                 //项目经理不同意，就不用提交给部门经理了，转向审核结束状态
                 request.setState(new  AuditOverState());
                 //继续执行下一步工作
                 request.doWork();
             }
         }
     }
}
```

接下来看看处理部门经理审核的状态类的实现。示例代码如下：

```
import java.util.Scanner;
/**
 * 处理部门经理的审核，处理后对应审核结束状态
 */
public class DepManagerState implements LeaveRequestState{
    public void doWork(StateMachine request) {
        //先把业务对象造型回来
        LeaveRequestModel lrm =
                    (LeaveRequestModel)request.getBusinessVO();
        System.out.println("部门经理审核中，请稍候......");
        //模拟用户处理界面，通过控制台来读取数据
        System.out.println(lrm.getUser()+"申请从"
            +lrm.getBeginDate()+"开始请假"+lrm.getLeaveDays()
            +"天,请部门经理审核(1为同意，2为不同意)：");
        //读取从控制台输入的数据
        Scanner scanner = new Scanner(System.in);
        if(scanner.hasNext()){
            int a = scanner.nextInt();
            //设置回到上下文中
```

```
            String result = "不同意";
            if(a==1){
                result = "同意";
            }
            lrm.setResult("部门经理审核结果: "+result);
            //部门经理审核以后，直接转向审核结束状态了
            request.setState(new AuditOverState());
            //继续执行下一步工作
            request.doWork();
        }
    }
}
```

再来看看处理审核结束的状态类的实现。示例代码如下：

```
public class AuditOverState implements LeaveRequestState{
    public void doWork(StateMachine request) {
        //先把业务对象造型回来
        LeaveRequestModel lrm =
                 (LeaveRequestModel)request.getBusinessVO();
        System.out.println(lrm.getUser()
          +"，你的请假申请已经审核结束，结果是: "+lrm.getResult());
    }
}
```

（2）万事俱备，可以写个客户端，来开始我们的流程之旅了。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建业务对象，并设置业务数据
        LeaveRequestModel lrm = new LeaveRequestModel();
        lrm.setUser("小李");
        lrm.setBeginDate("2010-02-08");
        lrm.setLeaveDays(5);

        //创建上下文对象
        LeaveRequestContext request = new LeaveRequestContext();
        //为上下文对象设置业务数据对象
        request.setBusinessVO(lrm);
        //配置上下文，作为开始的状态，以后就不管了
        request.setState(new ProjectManagerState());

        //请求上下文，让上下文开始处理工作
        request.doWork();
    }
}
```

辛苦了这么久，一定要好好地运行一下，体会在流程处理中是如何使用状态模式的。

第一步：运行一下，刚开始会出现如下信息：

```
项目经理审核中，请稍候......
小李申请从 2010-02-08 开始请假 5 天，请项目经理审核(1 为同意，2 为不同意)：
```

第二步：程序并没有停止，在等待你输入项目经理审核的结果，如果输入 1，表示同意，那么程序会继续判断，发现请假天数 5 天大于项目经理审核的范围了，会提交给部门经理审核。在控制台输入 1，然后回车看看，会出现如下信息：

```
项目经理审核中，请稍候......
小李申请从 2010-02-08 开始请假 5 天，请项目经理审核(1 为同意，2 为不同意)：
1
部门经理审核中，请稍候......
小李申请从 2010-02-08 开始请假 5 天，请部门经理审核(1 为同意，2 为不同意)：
```

第三步：同样，程序仍然没有停止，在等待你输入部门经理审核的结果，假如输入 1，然后回车，看看会发生什么。提示信息如下：

```
项目经理审核中，请稍候......
小李申请从 2010-02-08 开始请假 5 天，请项目经理审核(1 为同意，2 为不同意)：
1
部门经理审核中，请稍候......
小李申请从 2010-02-08 开始请假 5 天，请部门经理审核(1 为同意，2 为不同意)：
1
小李，你的请假申请已经审核结束，结果是：部门经理审核结果：同意
```

这个时候流程运行结束了，程序运行也结束了，有点流程控制的意味了吧。

如果上面第一步运行以后，在第二步输入 2，也就是项目经理不同意，会怎样呢？应该就不会再到部门经理了吧，试试看。运行提示信息如下：

```
项目经理审核中，请稍候......
小李申请从 2010-02-08 开始请假 5 天，请项目经理审核(1 为同意，2 为不同意)：
2
小李，你的请假申请已经审核结束，结果是：项目经理审核结果：不同意
```

（3）小结。

事实上，上面的程序可以和数据库结合起来，比如把审核结果存放在数据库中，也可以把审核的步骤也放到数据库中，每次运行的时候从数据库中获取这些值，判断是创建哪一个状态处理类，然后执行相应的处理就可以了。

现在这些东西都在内存中，所以程序不能停止，否则流程就运行不下去了。

另外，为了演示的简洁性，这里做了相应的简化，比如没有去根据申请人选择相应的项目经理和部门经理，也没有去考虑如果申请人就是项目经理或者部门经理怎么办。只是为了让大家看明白状态模式在其中的应用，主要是为了体现状态模式而不是业务。

18.3.5　状态模式的优缺点

状态模式有以下优点。

- 简化应用逻辑控制

  状态模式使用单独的类来封装一个状态的处理。如果把一个大的程序控制分成很多小块，每块定义一个状态来代表，那么就可以把这些逻辑控制的代码分散到很多单独的状态类中去，这样就把着眼点从执行状态提高到整个对象的状态，使得代码结构化和意图更清晰，从而简化应用的逻辑控制。

  对于依赖于状态的 if-else，理论上来讲，也可以改变成应用状态模式来实现，把每个 if 或 else 块定义一个状态来代表，那么就可以把块内的功能代码移动到状态处理类中，从而减少 if-else，避免出现巨大的条件语句。

- 更好地分离状态和行为

  状态模式通过设置所有状态类的公共接口，把状态和状态对应的行为分离开，把所有与一个特定的状态相关的行为都放入一个对象中，使得应用程序在控制的时候，只需要关心状态的切换，而不用关心这个状态对应的真正处理。

- 更好的扩展性

  引入了状态处理的公共接口后，使得扩展新的状态变得非常容易，只需要新增加一个实现状态处理的公共接口的实现类，然后在进行状态维护的地方，设置状态变化到这个新的状态即可。

- 显式化进行状态转换

  状态模式为不同的状态引入独立的对象，使得状态的转换变得更加明确。而且状态对象可以保证上下文不会发生内部状态不一致的情况，因为上下文中只有一个变量来记录状态对象，只要为这一个变量赋值就可以了。

状态模式也有一个很明显的缺点，一个状态对应一个状态处理类，会使得程序引入太多的状态类，这样程序变得杂乱。

18.3.6　思考状态模式

1. 状态模式的本质

> 状态模式的本质：**根据状态来分离和选择行为。**

仔细分析状态模式的结构，如果没有上下文，那么就退化回到只有接口和实现了，正是通过接口，把状态和状态对应的行为分开，才使得通过状态模式设计的程序易于扩展和维护。

而上下文主要负责的是公共的状态驱动，每当状态发生改变的时候，通常都是回调上下文来执行状态对应的功能。当然，上下文自身也可以维护状态的变化，另外，上下

文通常还会作为多个状态处理类之间的数据载体，在多个状态处理类之间传递数据。

2. 何时选用状态模式

建议在以下情况中选用状态模式。

- 如果一个对象的行为取决于它的状态，而且它必须在运行时刻根据状态来改变它的行为，可以使用状态模式，来把状态和行为分离开。虽然分离开了，但状态和行为是有对应关系的，可以在运行期间，通过改变状态，就能够调用到该状态对应的状态处理对象上去，从而改变对象的行为。
- 如果一个操作中含有庞大的多分支语句，而且这些分支依赖于该对象的状态，可以使用状态模式，把各个分支的处理分散包装到单独的对象处理类中，这样，这些分支对应的对象就可以不依赖于其他对象而独立变化了。

18.3.7 相关模式

- 状态模式和策略模式

  这是两个结构相同，功能各异的模式，具体的在策略模式里面讲过了，这里就不再赘述。
- 状态模式和观察者模式

  这两个模式乍一看，功能是很相似的，但是又有区别，可以组合使用。

  这两个模式都是在状态发生改变的时候触发行为，只不过观察者模式的行为是固定的，那就是通知所有的观察者；而状态模式是根据状态来选择不同的处理。

  从表面来看，两个模式功能相似，观察者模式中的被观察对象就好比状态模式中的上下文，观察者模式中当被观察对象的状态发生改变的时候，触发的通知所有观察者的方法就好比是状态模式中，根据状态的变化选择对应的状态处理。

  但实际这两个模式是不同的，观察者模式的目的是在被观察者的状态发生改变的时候，触发观察者联动，具体如何处理观察者模式不管；而状态模式的主要目的在于根据状态来分离和选择行为，当状态发生改变的时候，动态地改变行为。

  这两个模式是可以组合使用的，比如在观察者模式的观察者部分，当被观察对象的状态发生了改变，触发通知了所有的观察者以后，观察者该怎么处理呢？这个时候就可以使用状态模式，根据通知过来的状态选择相应的处理。
- 状态模式和单例模式

  这两个模式可以组合使用，可以把状态模式中的状态处理类实现成单例。

  状态模式和享元模式

  这两个模式可以组合使用。

  由于状态模式把状态对应的行为分散到多个状态对象中，会造成很多细粒度的状态对象，可以把这些状态处理对象通过享元模式来共享，从而节省资源。

第19章 备

(M

19.1 场景问题

19.1.1 开发仿真系统

考虑这样一个仿真应用，功能是，模拟运行针对某个具体问题的多个解决方案，记录运行过程的各种数据，在模拟运行完成以后，方便对这多个解决方案进行比较和评价，从而选定最优的解决方案。

这种仿真系统，在很多领域都有应用，比如工作流系统，对同一问题制定多个流程，然后通过仿真运行，最后来确定最优的流程作为解决方案；在工业设计和制造领域，仿真系统的应用就更广泛了。

由于都是解决同一个具体的问题，这多个解决方案并不是完全不一样的，假定它们的前半部分运行是完全一样的，只是在后半部分采用了不同的解决方案，后半部分需要使用前半部分运行所产生的数据。

由于要模拟运行多个解决方案，而且最后要根据运行结果来进行评价，这就意味着每个方案的后半部分的初始数据应该是一样的。也就是说在运行每个方案后半部分之前，要保证数据都是由前半部分运行所产生的数据。当然，这里并不具体地去深入到底有哪些解决方案，也不去深入到底有哪些状态数据，只是示意一下。

那么，这样的系统该如何实现呢？尤其是每个方案运行需要的初始数据应该一样，要如何来保证呢？

19.1.2 不用模式的解决方案

要保证初始数据的一致，实现思路也很简单：

（1）首先模拟运行流程第一个阶段，得到后阶段各个方案运行需要的数据，并把数据保存下来，以备后用。

（2）每次在模拟运行某一个方案之前，用保存的数据去重新设置模拟运行流程的对象，这样运行后面不同的方案时，对于这些方案，初始数据就是一样的了。

（1）根据上面的思路，写出仿真运行的示意代码。示例代码如下：

```
/**
 * 模拟运行流程A，只是一个示意，代指某个具体流程
 */
public class FlowAMock {
    /**
     * 流程名称，不需要外部存储的状态数据
     */
    private String flowName;
    /**
     * 示意，代指某个中间结果，需要外部存储的状态数据
```

```
     */
    private int tempResult;
    /**
     * 示意，代指某个中间结果，需要外部存储的状态数据
     */
    private String tempState;
    /**
     * 构造方法，传入流程名称
     * @param flowName 流程名称
     */
    public FlowAMock(String flowName){
        this.flowName = flowName;
    }

    public String getTempState() {
        return tempState;
    }
    public void setTempState(String tempState) {
        this.tempState = tempState;
    }
    public int getTempResult() {
        return tempResult;
    }
    public void setTempResult(int tempResult) {
        this.tempResult = tempResult;
    }

    /**
     * 示意，运行流程的第一个阶段
     */
    public void runPhaseOne(){
        //在这个阶段，可能产生了中间结果，示意一下
        tempResult = 3;
        tempState = "PhaseOne";
    }
    /**
     * 示意，按照方案一来运行流程后半部分
     */
    public void schema1(){
        //示意，需要使用第一个阶段产生的数据
```

```
        this.tempState += ",Schema1";
        System.out.println(this.tempState
                                + " : now run "+tempResult);
        this.tempResult += 11;
    }
    /**
     * 示意，按照方案二来运行流程后半部分
     */
    public void schema2(){
        //示意，需要使用第一个阶段产生的数据
        this.tempState += ",Schema2";
        System.out.println(this.tempState
                                + " : now run "+tempResult);
        this.tempResult += 22;
    }
}
```

（2）看看如何使用这个模拟流程的对象，写个客户端来测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        // 创建模拟运行流程的对象
        FlowAMock mock = new FlowAMock("TestFlow");
        //运行流程的第一个阶段
        mock.runPhaseOne();
        //得到第一个阶段运行所产生的数据，后面要用
        int tempResult = mock.getTempResult();
        String tempState = mock.getTempState();

        //按照方案一来运行流程后半部分
        mock.schema1();

        //把第一个阶段运行所产生的数据重新设置回去
        mock.setTempResult(tempResult);
        mock.setTempState(tempState);

        //按照方案二来运行流程后半部分
        mock.schema2();
    }
}
```

运行结果如下：

```
PhaseOne,Schema1 : now run 3
PhaseOne,Schema2 : now run 3
```

仔细看上面结果中框住的部分，是一样的值，这说明，运行时它们的初始数据是一样的，基本满足了功能要求。

19.1.3 有何问题

看起来实现很简单，是吧，想一想有没有什么问题呢？

上面的实现有一个不太理想的地方，那就是数据是一个一个零散着在外部存放的，如果需要外部存放的数据多了，会显得很杂乱。这个容易解决，只需要定义一个数据对象来封装这些需要外部存放的数据就可以了。上面那样做是有意的，好提醒大家这个问题。这个就不再示例了。

还有一个严重的问题，那就是，为了把运行期间的数据放到外部存储起来，模拟流程的对象被迫把内部数据结构开放出来，这暴露了对象的实现细节，而且也破坏了对象的封装性。本来这些数据只是模拟流程的对象内部数据，应该是不对外的。

那么究竟如何实现这样的功能会比较好呢？

19.2 解 决 方 案

19.2.1 使用备忘录模式来解决问题

来解决上述问题的一个合理的解决方案就是备忘录模式。那么什么是备忘录模式呢？

1. 备忘录模式的定义

> 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

一个备忘录是一个对象，它存储另一个对象在某个瞬间的内部状态，后者被称为备忘录的原发器。

2. 应用备忘录模式来解决问题的思路

仔细分析上面的示例功能，需要在运行期间捕获模拟流程运行对象的内部状态。这些需要捕获的内部状态就是它运行第一个阶段产生的内部数据，并且在该对象之外来保存这些状态，因为在后面它有不同的运行方案。但是这些不同的运行方案需要的初始数据是一样的，都是流程在第一个阶段运行所产生的数据，这就要求运行每个方案后半部分前，要把该对象的状态恢复到第一个阶段运行结束时的状态。

在这个示例中出现的、需要解决的问题就是：如何能够在不破坏对象的封装性的前提下，来保存和恢复对象的状态。

看起来跟备忘录模式要解决的问题是如此的贴切，备忘录模式简直像是专为这个应用打造的一样。那么使用备忘录模式如何来解决这个问题呢？

备忘录模式引入一个存储状态的备忘录对象，为了让外部无法访问这个对象的值，一般把这个对象实现成为需要保存数据的对象的内部类，通常还是私有的。这样一来，除了这个需要保存数据的对象，外部无法访问到这个备忘录对象的数据，这就保证了对象的封装性不被破坏。

但是这个备忘录对象需要存储在外部。为了避免让外部访问到这个对象内部的数据，备忘录模式引入了一个备忘录对象的窄接口，这个接口一般是空的，什么方法都没有，这样外部存储的地方，只是知道存储了一些备忘录接口的对象，但是由于接口是空的，它们无法通过接口去访问备忘录对象内部的数据。

19.2.2 备忘录模式的结构和说明

备忘录模式的结构如图 19.1 所示。

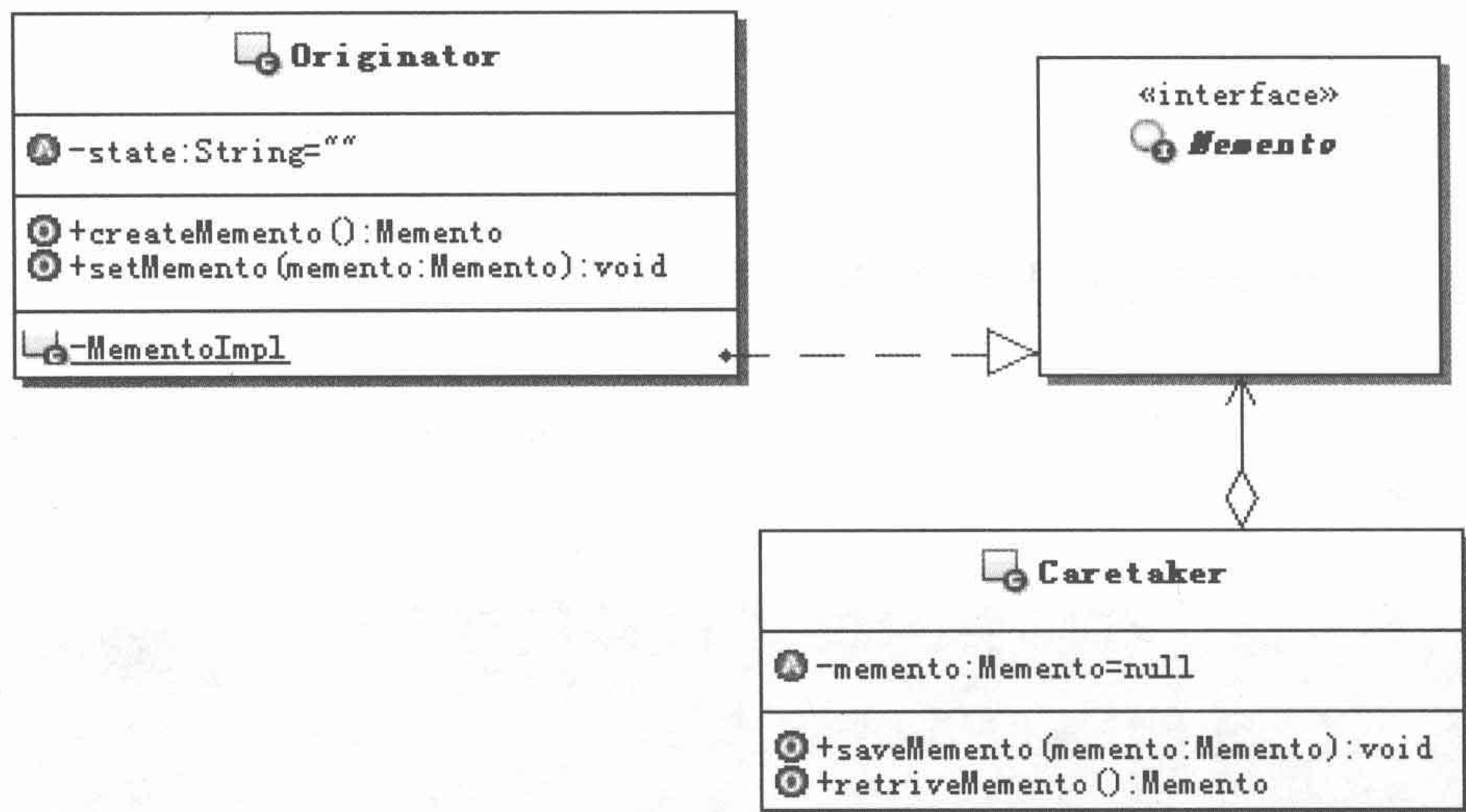


图 19.1 备忘录模式的结构示意图

- Memento：备忘录。主要用来存储原发器对象的内部状态，但是具体需要存储哪些数据是由原发器对象来决定的。另外备忘录应该只能由原发器对象来访问它内部的数据，原发器外部的对象不应该访问到备忘录对象的内部数据。
- Originator：原发器。使用备忘录来保存某个时刻原发器自身的状态，也可以使用备忘录来恢复内部状态。
- Caretaker：备忘录管理者，或者称为备忘录负责人。主要负责保存备忘录对象，但是不能对备忘录对象的内容进行操作或检查。

19.2.3 备忘录模式示例代码

（1）先看看备忘录对象的窄接口，就是那个 Memento 接口，这个实现最简单，是个空的接口，没有任何方法定义。示例代码如下：

```
/**
 * 备忘录的窄接口，没有任何方法定义
 */
public interface Memento {
    //
}
```

（2）再看看原发器对象，它里面会有备忘录对象的实现，因为真正的备忘录对象当作原发器对象的一个私有内部类来实现了。示例代码如下：

```
/**
 * 原发器对象
 */
public class Originator {
    /**
     * 示意，表示原发器的状态
     */
    private String state = "";
    /**
     * 创建保存原发器对象的状态的备忘录对象
     * @return 创建好的备忘录对象
     */
    public Memento createMemento() {
        return new MementoImpl(state);
    }
    /**
     * 重新设置原发器对象的状态，让其回到备忘录对象记录的状态
     * @param memento 记录有原发器状态的备忘录对象
     */
    public void setMemento(Memento memento) {
        MementoImpl mementoImpl = (MementoImpl)memento;
        this.state = mementoImpl.getState();
    }
    /**
     * 真正的备忘录对象，实现备忘录窄接口
     * 实现成私有的内部类，不让外部访问
     */
```

```
    private static class MementoImpl implements Memento{
        /**
         * 示意，表示需要保存的状态
         */
        private String state = "";
        public MementoImpl(String state){
            this.state = state;
        }
        public String getState() {
            return state;
        }
    }
}
```

创建过后，一般只让外面来访问数据，而不再修改数据，因此只有 getter

（3）接下来看看备忘录管理者对象。示例代码如下：

```
/**
 * 负责保存备忘录的对象
 */
public class Caretaker{
    /**
     * 记录被保存的备忘录对象
     */
    private Memento memento = null;
    /**
     * 保存备忘录对象
     * @param memento 被保存的备忘录对象
     */
    public void saveMemento(Memento memento){
        this.memento = memento;
    }
    /**
     * 获取被保存的备忘录对象
     * @return 被保存的备忘录对象
     */
    public Memento retriveMemento(){
        return this.memento;
    }
}
```

19.2.4　使用备忘录模式重写示例

学习了备忘录模式的基本知识以后，尝试一下使用备忘录模式把前面的示例重写，来看看如何使用备忘录模式。

（1）首先，那个模拟流程运行的对象就相当于备忘录模式中的原发器；

（2）而它要保存的数据，原来是零散的，现在做一个备忘录对象来存储这些数据，并且把这个备忘录对象实现成为内部类；

（3）当然为了保存这个备忘录对象，还是需要提供管理者对象的；

（4）为了和管理者对象交互，管理者需要知道保存对象的类型，那就提供一个备忘录对象的窄接口来供管理者使用，相当于标识了类型。

此时程序的结构如图 19.2 所示。

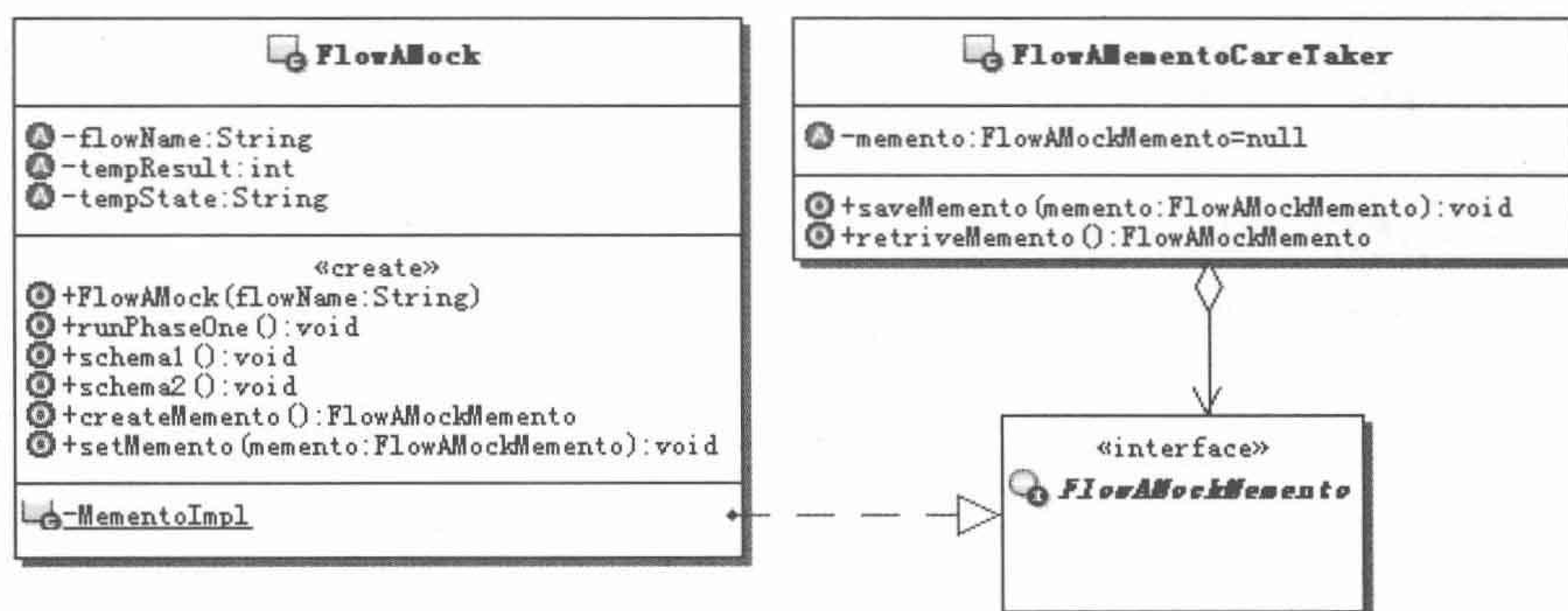


图 19.2　使用备忘录模式重写示例的结构示意图

（1）先来看看备忘录对象的窄接口。示例代码如下：

```
/**
 * 模拟运行流程 A 的对象的备忘录接口，是个窄接口
 */
public interface FlowAMockMemento {
    //空的
}
```

（2）再来看看新的模拟运行流程 A 的对象，相当于原发器对象了，它的变化比较多，大致有如下变化。

- 这个对象原来暴露出去的内部状态，不用再暴露出去了，也就是内部状态不用再对外提供 getter/setter 方法了。
- 在这个对象中提供一个私有的备忘录对象，里面封装想要保存的内部状态，同时让这个备忘录对象实现备忘录对象的窄接口。
- 在这个对象中提供创建备忘录对象和根据备忘录对象恢复内部状态的方法。

具体的示例代码如下：

```
/**
 * 模拟运行流程 A，只是一个示意，代指某个具体流程
```

```
*/
public class FlowAMock {
    /**
     * 流程名称，不需要外部存储的状态数据
     */
    private String flowName;
    /**
     * 示意，代指某个中间结果，需要外部存储的状态数据
     */
    private int tempResult;
    /**
     * 示意，代指某个中间结果，需要外部存储的状态数据
     */
    private String tempState;
    /**
     * 构造方法，传入流程名称
     * @param flowName 流程名称
     */
    public FlowAMock(String flowName){
        this.flowName = flowName;
    }
    /**
     * 示意，运行流程的第一个阶段
     */
    public void runPhaseOne(){
        //在这个阶段，可能产生了中间结果，示意一下
        tempResult = 3;
        tempState = "PhaseOne";
    }
    /**
     * 示意，按照方案一来运行流程后半部分
     */
    public void schema1(){
        //示意，需要使用第一个阶段产生的数据
        this.tempState += ",Schema1";
        System.out.println(this.tempState
                                    + " : now run "+tempResult);
        this.tempResult += 11;
    }
    /**
```

```
     * 示意，按照方案二来运行流程后半部分
     */
    public void schema2(){
        //示意，需要使用第一个阶段产生的数据
        this.tempState += ",Schema2";
        System.out.println(this.tempState
                                  + " : now run "+tempResult);
        this.tempResult += 22;
    }
    /**
     * 创建保存原发器对象状态的备忘录对象
     * @return 创建好的备忘录对象
     */
    public FlowAMockMemento createMemento() {
        return new MementoImpl(this.tempResult,this.tempState);
    }
    /**
     * 重新设置原发器对象的状态，让其回到备忘录对象记录的状态
     * @param memento 记录有原发器状态的备忘录对象
     */
    public void setMemento(FlowAMockMemento memento) {
        MementoImpl mementoImpl = (MementoImpl)memento;
        this.tempResult = mementoImpl.getTempResult();
        this.tempState = mementoImpl.getTempState();
    }
    /**
     * 真正的备忘录对象，实现备忘录窄接口
     * 实现成私有的内部类，不让外部访问
     */
    private static class MementoImpl implements FlowAMockMemento{
        /**
         * 示意，保存某个中间结果
         */
        private int tempResult;
        /**
         * 示意，保存某个中间结果
         */
        private String tempState;
        public MementoImpl(int tempResult,String tempState){
            this.tempResult = tempResult;
```

```
            this.tempState = tempState;
        }
        public int getTempResult() {
            return tempResult;
        }
        public String getTempState() {
            return tempState;
        }
    }
}
```

（3）接下来要来实现提供保存备忘录对象的管理者了。示例代码如下：

```
/**
 * 负责保存模拟运行流程 A 的对象的备忘录对象
 */
public class FlowAMementoCareTaker {
    /**
     * 记录被保存的备忘录对象
     */
    private FlowAMockMemento memento = null;
    /**
     * 保存备忘录对象
     * @param memento 被保存的备忘录对象
     */
    public void saveMemento(FlowAMockMemento memento){
        this.memento = memento;
    }
    /**
     * 获取被保存的备忘录对象
     * @return 被保存的备忘录对象
     */
    public FlowAMockMemento retriveMemento(){
        return this.memento;
    }
}
```

（4）最后来看看，如何使用上面按照备忘录模式实现的这些对象呢？写个新的客户端来测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        // 创建模拟运行流程的对象
```

```
        FlowAMock mock = new FlowAMock("TestFlow");
        //运行流程的第一个阶段
        mock.runPhaseOne();
        //创建一个管理者
        FlowAMementoCareTaker careTaker =
                                   new FlowAMementoCareTaker();
        //创建此时对象的备忘录对象，并保存到管理者对象那里，后面要用
        FlowAMockMemento memento = mock.createMemento();
        careTaker.saveMemento(memento);

        //按照方案一来运行流程的后半部分
        mock.schema1();

        //从管理者获取备忘录对象，然后设置回去
        //让模拟运行流程的对象自己恢复自己的内部状态
        mock.setMemento(careTaker.retriveMemento());

        //按照方案二来运行流程的后半部分
        mock.schema2();
    }
}
```

运行结果和前面的示例是一样的。结果如下：

```
PhaseOne,Schema1 : now run 3
PhaseOne,Schema2 : now run 3
```

好好体会一下上面的示例，由于备忘录对象是一个私有的内部类，外面只能通过备忘录对象的窄接口来获取备忘录对象，而这个接口没有任何方法，仅仅起到了一个标识对象类型的作用，从而保证内部的数据不会被外部获取或是操作，保证了原发器对象的封装性，也就不再暴露原发器对象的内部结构了。

19.3　模 式 讲 解

19.3.1　认识备忘录模式

1. 备忘录模式的功能

备忘录模式的功能，首先是在不破坏封装性的前提下，捕获一个对象的内部状态。这里要注意两点，一个是不破坏封装性，也就是对象不能暴露它不应该暴露的细节；另外一个是捕获的是对象的内部状态，而且通常还是运行期间某个时刻对象的内部状态。

为什么要捕获这个对象的内部状态呢？捕获这个内部状态有什么用呢？

是为了在以后的某个时候，将该对象的状态恢复到备忘录所保存的状态，这才是备忘录真正的目的。前面保存状态就是为了后面恢复，虽然不是一定要恢复，但是目的是为了恢复。这也是很多人理解备忘录模式的时候，忽视掉的地方，他们太关注备忘，而忽视了恢复，这是不全面的理解。

捕获的状态存放在哪里呢？

备忘录模式中，捕获的内部状态存储在备忘录对象中；而备忘录对象通常会被存储在原发器对象之外，也就是被保存状态的对象的外部，通常是存放在管理者对象那里。

2. 备忘录对象

在备忘录模式中，备忘录对象通常就是用来记录原发器需要保存的状态的对象，简单点的实现，也就是封装数据的对象。

但是备忘录对象和普通的封装数据的对象还是有区别的，主要是备忘录对象一般只让原发器对象来操作，而不是像普通的封装数据的对象那样，谁都可以使用。为了保证这一点，通常会把备忘录对象作为原发器对象的内部类来实现，而且实现成私有的，这就断绝了外部来访问这个备忘录对象的途径。

备忘录对象需要保存在原发器对象之外，为了与外部交互，通常备忘录对象都会实现一个窄接口，来标识对象的类型。

3. 原发器对象

原发器对象就是需要被保存状态的对象，也是有可能需要恢复状态的对象。原发器一般会包含备忘录对象的实现。

通常原发器对象应该提供捕获某个时刻对象内部状态的方法，在这个方法中，原发器对象会创建备忘录对象，把需要保存的状态数据设置到备忘录对象中，然后把备忘录对象提供给管理者对象来保存。

当然，原发器对象也应该提供这样的方法：按照外部要求来恢复内部状态到某个备忘录对象记录的状态。

4. 管理者对象

在备忘录模式中，管理者对象主要是负责保存备忘录对象。这里有几点要讲一下。

- 并不一定要特别的做出一个管理者对象来。广义地说，调用原发器获得备忘录对象后，备忘录对象放在哪里，哪个对象就可以算是管理者对象。
- 管理者对象并不是只能管理一个备忘录对象，一个管理者对象可以管理很多的备忘录对象。虽然前面的示例中是保存一个备忘录对象，但别忘了那只是个示意，并不是只能实现成那样。
- 狭义的管理者对象是只管理同一类的备忘录对象，但广义的管理者对象是可以管理不同类型的备忘录对象的。
- 管理者对象需要实现的基本功能主要是：存入备忘录对象、保存备忘录对象和获取备忘录对象。如果从功能上看，就是一个缓存功能的实现，或者是一个简单的对象实例池的实现。
- 管理者虽然能存取备忘录对象，但是不能访问备忘录对象内部的数据。

5. 窄接口和宽接口

在备忘录模式中，为了控制对备忘录对象的访问，出现了窄接口和宽接口的概念。

- 窄接口：管理者只能看到备忘录的窄接口，窄接口的实现中通常没有任何的方法，只是一个类型标识。窄接口使得管理者只能将备忘录传递给其他对象。
- 宽接口：原发器能够看到一个宽接口，允许它访问所需的所有数据，来返回到先前的状态。理想状况是：只允许生成备忘录的原发器来访问该备忘录的内部状态，通常实现成为原发器内的一个私有内部类。

在前面的示例中，定义了一个名称为 FlowAMockMemento 的接口，里面没有定义任何方法，然后让备忘录来实现这个接口，从而标识备忘录就是这么一个 FlowAMockMemento 的类型，这个接口就是窄接口。

在前面的实现中，备忘录对象是实现在原发器内的一个私有内部类，只有原发器对象能访问它，原发器可以访问到备忘录对象中所有的内部状态，这就是宽接口。

这也算是备忘录模式的标准实现方式，那就是窄接口没有任何的方法，把备忘录对象实现成为原发器对象的私有内部类。

> 注意
>
> **那么能不能在窄接口中提供备忘录对象对外的方法，变相对外提供一个“宽”点的接口呢？**
>
> **通常情况是不会这么做的。因为这样一来，所有能拿到这个接口的对象就可以通过这个接口来访问备忘录内部的数据或是功能，这违反了备忘录模式的初衷，备忘录模式要求“在不破坏封装性的前提下”，如果这么做，那就等于是暴露了内部细节。因此，备忘录模式在实现的时候，对外多是采用窄接口，而且通常不会定义任何方法。**

6. 使用备忘录的潜在代价

标准的备忘录模式的实现机制是依靠缓存来实现的，因此，当需要备忘的数据量较大时，或者是存储的备忘录对象数据量不大但是数量很多的时候，或者是用户很频繁地创建备忘录对象的时候，这些都会导致非常大的开销。

因此在使用备忘录模式的时候，一定要好好思考应用的环境，如果使用的代价太高，就不要选用备忘录模式，可以采用其他的替代方案。

7. 增量存储

如果需要频繁地创建备忘录对象，而且创建和应用备忘录对象来恢复状态的顺序是可控的，那么可以让备忘录进行增量存储，也就是备忘录可以仅仅存储原发器内部相对于上一次存储状态后的增量改变。

比如，在命令模式实现可撤销命令的实现中，就可以使用备忘录来保存每个命令对应的状态，然后在撤销命令的时候，使用备忘录来恢复这些状态。由于命令的历史列表是按照命令操作的顺序来存放的，也是按照这个历史列表来进行取消和重做的，因此顺序是可控的。那么这种情况，还可以让备忘录对象只存储一个命令所产生的增量改变而不是它所影响的每一个对象的完整状态。

8. 备忘录模式调用顺序示意图

在使用备忘录模式的时候，分成了两个阶段，第一个阶段是创建备忘录对象的阶段，第二个阶段是使用备忘录对象来恢复原发器对象状态的阶段。它们的调用顺序是不一样的，下面分别用图来示意一下。

先看看创建备忘录对象的阶段。调用顺序如图 19.3 所示。

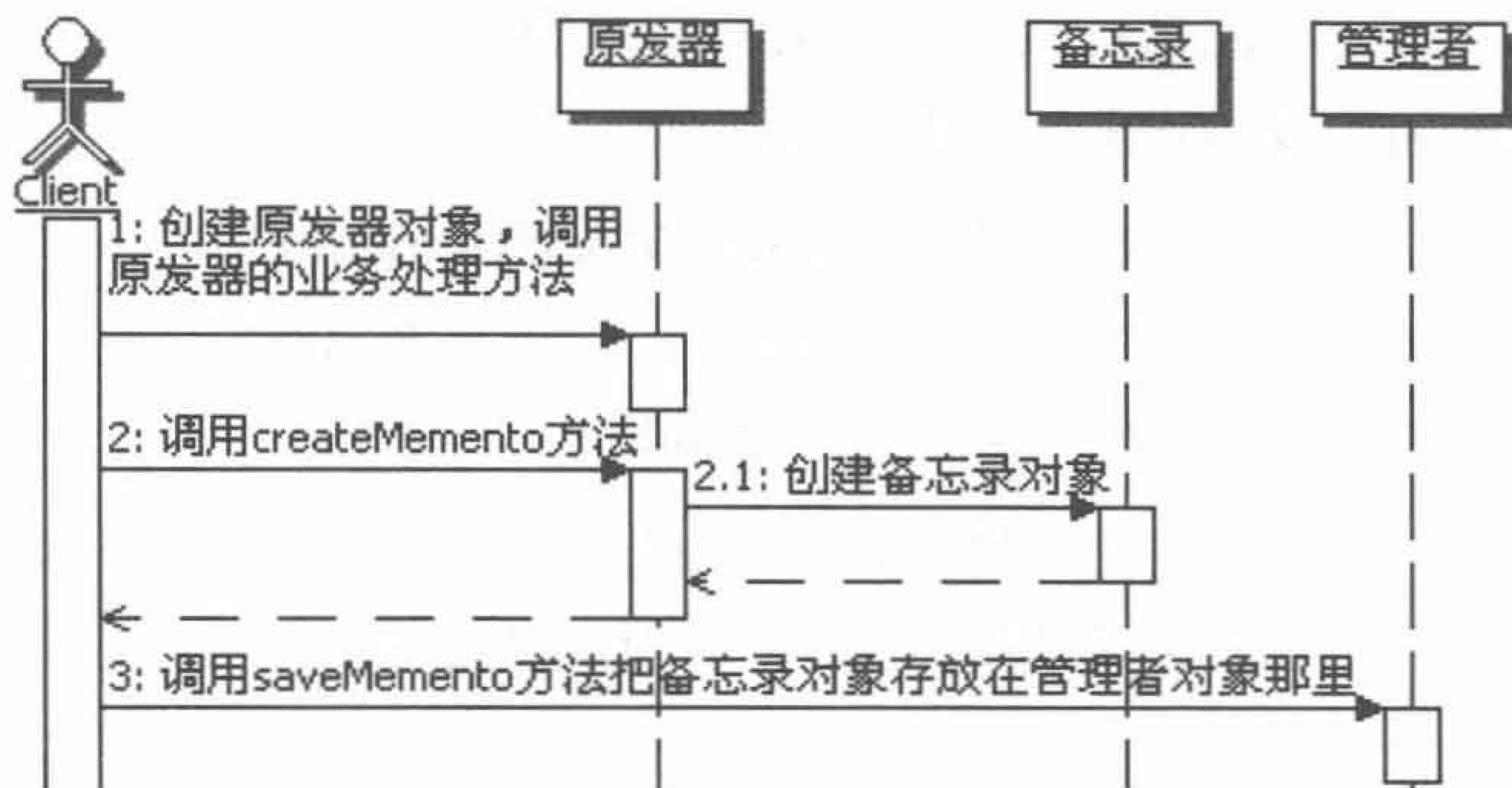


图 19.3　创建备忘录对象的调用顺序示意图

再看看使用备忘录对象来恢复原发器对象状态的阶段。调用顺序如图 19.4 所示。



图 19.4　使用备忘录对象来恢复原发器对象状态的调用顺序示意图

19.3.2　结合原型模式

在原发器对象创建备忘录对象的时候，如果原发器对象中全部或者大部分的状态都需要保存，一个简洁的方式就是直接克隆一个原发器对象。也就是说，这个时候备忘录对象中存放的是一个原发器对象的实例。

还是通过示例来说明。只需要修改原发器对象就可以了，大致有如下变化。

- 原发器对象要实现可克隆的，好在这个原发器对象的状态数据都很简单，都是基本数据类型，所以直接使用默认的克隆方法就可以了，不用自己实现克隆，更不涉及深度克隆，否则，正确实现深度克隆还是个问题。
- 备忘录对象的实现要修改，只需要存储原发器对象克隆出来的实例对象就可以了。
- 相应的创建和设置备忘录对象的地方都要做修改。

示例代码如下：

```
/**
 * 模拟运行流程 A，只是一个示意，代指某个具体流程
```

```
 */
public class FlowAMockPrototype implements Cloneable {
    private String flowName;
    private int tempResult;
    private String tempState;
    public FlowAMockPrototype(String flowName){
        this.flowName = flowName;
    }

    public void runPhaseOne(){
        //在这个阶段，可能产生了中间结果，示意一下
        tempResult = 3;
        tempState = "PhaseOne";
    }
    public void schema1(){
        //示意，需要使用第一个阶段产生的数据
        this.tempState += ",Schema1";
        System.out.println(this.tempState
                        + " : now run "+tempResult);
        this.tempResult += 11;
    }

    public void schema2(){
        //示意，需要使用第一个阶段产生的数据
        this.tempState += ",Schema2";
        System.out.println(this.tempState
                        + " : now run "+tempResult);
        this.tempResult += 22;
    }
    /**
     * 创建保存原发器对象状态的备忘录对象
     * @return 创建好的备忘录对象
     */
    public FlowAMockMemento createMemento() {
        try {
            return new MementoImplPrototype(
                        (FlowAMockPrototype) this.clone());
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
```

这些都没有改变

```
        return null;
    }
    /**
     * 重新设置原发器对象的状态，让其回到备忘录对象记录的状态
     * @param memento 记录有原发器状态的备忘录对象
     */
    public void setMemento(FlowAMockMemento memento) {
        MementoImplPrototype mementoImpl =
                                    (MementoImplPrototype)memento;
        this.tempResult = mementoImpl.getFlowAMock().tempResult;
        this.tempState = mementoImpl.getFlowAMock().tempState;
    }
    /**
     * 真正的备忘录对象，实现备忘录窄接口，实现成私有的内部类，不让外部访问
     */
    private static class MementoImplPrototype
                                implements FlowAMockMemento{
        private FlowAMockPrototype flowAMock = null;

        public MementoImplPrototype(FlowAMockPrototype f){
            this.flowAMock = f;
        }

        public FlowAMockPrototype getFlowAMock() {
            return flowAMock;
        }
    }
}
```

好了，结合原型模式来实现备忘录模式的示例就写好了，在前面的客户测试程序中，创建原发器对象的时候，使用这个新实现的原发器对象就可以了。去测试和体会一下，看看是否能正确地实现需要的功能。

> 注意：**不过要注意一点，就是如果克隆对象非常复杂，或者需要很多层次的深度克隆，实现克隆的时候会比较麻烦。**

19.3.3 离线存储

标准的备忘录模式，没有讨论离线存储的实现。

事实上，从备忘录模式的功能和实现上，是可以把备忘录的数据实现成为离线存储

的，也就是不仅限于存储在内存中，可以把这些备忘数据存储到文件中、XML 中、数据库中，从而支持跨越会话的备份和恢复功能。

离线存储甚至能帮助应对应用崩溃，然后关闭重启的情况。应用重启后，从离线存储中获取相应的数据，然后重新设置状态，恢复到崩溃前的状态。

当然，并不是所有的备忘数据都需要离线存储。一般来讲，需要存储很长时间，或者需要支持跨越会话的备份和恢复功能，或者是希望系统关闭后还能被保存的备忘数据，这些情况建议采用离线存储。

离线存储的实现也很简单，就以前面模拟运行流程的应用来说，如果要实现离线存储，主要需要修改管理者对象，把它保存备忘录对象的方法实现成为保存到文件中，而恢复备忘录对象实现成为读取文件就可以了。对于其他的相关对象，主要是要实现序列化，只有可序列化的对象才能被存储到文件中。

如果实现保存备忘录对象到文件，就不用在内存中保存了，删除用来“记录被保存的备忘录对象”的这个属性。示例代码如下：

```
/**
 * 负责在文件中保存模拟运行流程 A 的对象的备忘录对象
 */
public class FlowAMementoFileCareTaker {
    /**
     * 保存备忘录对象
     * @param memento 被保存的备忘录对象
     */
    public void saveMemento(FlowAMockMemento memento){
        //写到文件中
        ObjectOutputStream out = null;
        try{
            out = new ObjectOutputStream(
                    new BufferedOutputStream(
                            new FileOutputStream("FlowAMemento")
                    )
            );
            out.writeObject(memento);
        }catch(Exception err){
            err.printStackTrace();
        }finally{
            try {
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
        }
    }
    /**
     * 获取被保存的备忘录对象
     * @return 被保存的备忘录对象
     */
    public FlowAMockMemento retriveMemento(){
        FlowAMockMemento memento = null;
        //从文件中获取备忘录数据
        ObjectInputStream in = null;
        try{
            in = new ObjectInputStream(
                    new BufferedInputStream(
                            new FileInputStream("FlowAMemento")
                    )
            );
            memento = (FlowAMockMemento)in.readObject();
        }catch(Exception err){
            err.printStackTrace();
        }finally{
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return memento;
    }
}
```

同时需要让备忘录对象的窄接口继承可序列化接口。示例代码如下：

```
/**
 * 模拟运行流程 A 的对象的备忘录接口，是个窄接口
 */
public interface FlowAMockMemento extends Serializable  {
}
```

还有 FlowAMock 对象，也需要实现可序列化。示例代码如下：

```
/**
 * 模拟运行流程 A，只是一个示意，代指某个具体流程
 */
```

```
public class FlowAMock implements Serializable  {
    //中间的实现省略了
}
```

好了，保存到文件的存储就实现好了。在前面的客户测试程序中，创建管理者对象的时候，使用这个新实现的管理者对象就可以了。去测试和体会一下。

19.3.4　再次实现可撤销操作

在命令模式中，讲到了可撤销的操作，在那里讲到：有两种基本的思路来实现可撤销的操作，一种是补偿式或者反操作式，比如被撤销的操作是加的功能，那撤销的实现就变成减的功能；同理被撤销的操作是打开的功能，那么撤销的实现就变成关闭的功能。

另外一种方式是存储恢复式，意思就是把操作前的状态记录下来，然后要撤销操作的时候就直接恢复回去就可以了。

这里就来实现第二种方式存储恢复式。为了让大家更好地理解可撤销操作的功能，还是用原来的那个例子，对比学习会比较清楚。

这也相当于是命令模式和备忘录模式结合的一个例子，而且由于命令列表的存在，对应保存的备忘录对象也有多个。

1. 范例需求

考虑一个计算器的功能，最简单的那种，只能实现加减法运算，现在要让这个计算器支持可撤销的操作。

2. 存储恢复式的解决方案

存储恢复式的实现，可以使用备忘录模式，大致实现的思路如下。

- 把原来的运算类，就是 Operation 类，当作原发器，原来的内部状态 result，就只提供一个 getter 方法，来让外部获取运算的结果。
- 在这个原发器中，实现一个私有的备忘录对象。
- 把原来的计算器类，就是 Calculator 类，当作管理者，把命令对应的备忘录对象保存在这里。当需要撤销操作的时候，就把相应的备忘录对象设置回到原发器中，恢复原发器的状态。

一起来看看具体的实现，会更清楚。

（1）定义备忘录对象的窄接口。示例代码如下：

```
public interface Memento {
    //空的
}
```

（2）定义命令的接口。有以下几点修改。

- 修改原来的 undo 方法，传入备忘录对象。
- 添加一个 redo 方法，传入备忘录对象。
- 添加一个 createMemento 的方法，获取需要被保存的备忘录对象。

示例代码如下：

```
/**
 * 定义一个命令的接口
 */
public interface Command {
    /**
     * 执行命令
     */
    public void execute();
    /**
     * 撤销命令，恢复到备忘录对象记录的状态
     * @param m 备忘录对象
     */
    public void undo(Memento m);
    /**
     * 重做命令，恢复到备忘录对象记录的状态
     * @param m 备忘录对象
     */
    public void redo(Memento m);
    /**
     * 创建保存原发器对象状态的备忘录对象
     * @return 创建好的备忘录对象
     */
    public Memento createMemento();
}
```

（3）再来定义操作运算的接口，相当于计算器类这个原发器对外提供的接口，它需要做如下的调整。

- 删除原有的 setResult 方法，内部状态，不允许外部操作。
- 添加一个 createMemento 的方法，获取需要保存的备忘录对象。
- 添加一个 setMemento 的方法，来重新设置原发器对象的状态。

示例代码如下：

```
/**
 * 操作运算的接口
 */
public interface OperationApi {
    /**
     * 获取计算完成后的结果
     * @return 计算完成后的结果
     */
```

```
    public int getResult();
    /**
     * 执行加法
     * @param num 需要加的数
     */
    public void add(int num);
    /**
     * 执行减法
     * @param num 需要减的数
     */
    public void substract(int num);
    /**
     * 创建保存原发器对象状态的备忘录对象
     * @return 创建好的备忘录对象
     */
    public Memento createMemento();
    /**
     * 重新设置原发器对象状态，让其回到备忘录对象记录的状态
     * @param memento 记录有原发器状态的备忘录对象
     */
    public void setMemento(Memento memento);
}
```

（4）由于现在撤销和恢复操作是通过使用备忘录对象，直接来恢复原发器的状态，因此不再需要按照操作类型来区分了，对于所有的命令实现，它们的撤销和重做都是一样的。原来的实现是要区分的，如果是撤销加的操作，那就是减，而撤销减的操作，那就是加。现在就不再区分了，统一使用备忘录对象来恢复。

因此，实现一个所有命令的公共对象，在其中把公共功能都实现了，这样每个命令在实现的时候就简单了。顺便把设置持有者的公共实现也放到这个公共对象中来，这样各个命令对象就不用再实现这个方法了。示例代码如下：

```
/**
 * 命令对象的公共对象，实现各个命令对象的公共方法
 */
public abstract class AbstractCommand implements Command{
    /**
     * 具体的功能实现，这里不管
     */
    public abstract void execute();
    /**
     * 持有真正的命令实现者对象
```

```
     */
    protected OperationApi operation = null;
    public void setOperation(OperationApi operation) {
        this.operation = operation;
    }
    public Memento createMemento() {
        return this.operation.createMemento();
    }
    public void redo(Memento m) {
        this.operation.setMemento(m);
    }
    public void undo(Memento m) {
        this.operation.setMemento(m);
    }
}
```

（5）有了公共的命令实现对象，各个具体命令的实现就简单了。实现加法命令的对象实现，不再直接实现 Command 接口了，而是继承命令的公共对象，这样只需要实现和自己命令相关的业务方法就可以了。示例代码如下：

```
public class AddCommand extends AbstractCommand{
    private int opeNum;
    public AddCommand(int opeNum){
        this.opeNum = opeNum;
    }
    public void execute() {
        this.operation.add(opeNum);
    }
}
```

看看减法命令的实现，跟加法命令的实现差不多。示例代码如下：

```
public class SubstractCommand extends AbstractCommand{
    private int opeNum;
    public SubstractCommand(int opeNum){
        this.opeNum = opeNum;
    }
    public void execute() {
        this.operation.substract(opeNum);
    }
}
```

（6）接下来看看运算类的实现，相当于是原发器对象，它的实现有如下改变。

- 不再提供 setResult 方法，内部状态，不允许外部来操作。

- 添加了 createMemento 和 setMemento 方法的实现。
- 添加实现了一个私有的备忘录对象。

示例代码如下：

```
/**
 * 运算类，真正实现加减法运算
 */
public class Operation implements OperationApi{
    /**
     * 记录运算的结果
     */
    private int result;
    public int getResult() {
        return result;
    }
    public void add(int num){
        result += num;
    }
    public void substract(int num){
        result -= num;
    }
    public Memento createMemento() {
        MementoImpl m = new MementoImpl(result);
        return m;
    }
    public void setMemento(Memento memento) {
        MementoImpl m = (MementoImpl)memento;
        this.result = m.getResult();
    }
    /**
     * 备忘录对象
     */
    private static class MementoImpl implements Memento{
        private int result = 0;
        public MementoImpl(int result){
            this.result = result;
        }
        public int getResult() {
            return result;
        }
    }
```

```
}
```

（7）接下来该看看如何具体地使用备忘录对象来实现撤销操作和重做操作了。同样在计算器类中实现，这个时候，计算器类就相当于是备忘录模式管理者对象。

> 提示
>
> **实现思路：由于对于每个命令对象，撤销和重做的状态是不一样的，撤销是回到命令操作前的状态，而重做是回到命令操作后的状态，因此对每一个命令，使用一个备忘录对象的数组来记录对应的状态。**
>
> **这些备忘录对象和命令对象是相对应的，因此也跟命令历史记录一样，设置相应的历史记录，它的顺序和命令完全对应起来。在操作命令历史记录的同时，对应操作相应的备忘录对象记录。**

示例代码如下：

```
/**
 * 计算器类，计算器上有加法按钮、减法按钮，还有撤销和恢复的按钮
 */
public class Calculator {
    /**
     * 命令的操作历史记录，在撤销时用
     */
    private List<Command> undoCmds = new ArrayList<Command>();
    /**
     * 命令被撤销的历史记录，在恢复时用
     */
    private List<Command> redoCmds = new ArrayList<Command>();
    /**
     * 命令操作对应的备忘录对象的历史记录，在撤销时用
     * 数组有两个元素，第一个是命令执行前的状态，第二个是命令执行后的状态
     */
    private List<Memento[]> undoMementos =
                                new ArrayList<Memento[]>();
    /**
     * 被撤销命令对应的备忘录对象的历史记录，在恢复时用
     * 数组有两个元素，第一个是命令执行前的状态，第二个是命令执行后的状态
     */
    private List<Memento[]> redoMementos =
                                    new ArrayList<Memento[]>();

    private Command addCmd = null;
    private Command substractCmd = null;
    public void setAddCmd(Command addCmd) {
```

```
        this.addCmd = addCmd;
    }
    public void setSubstractCmd(Command substractCmd) {
        this.substractCmd = substractCmd;
    }

    public void addPressed(){
        //获取对应的备忘录对象，并保存在相应的历史记录中
        Memento m1 = this.addCmd.createMemento();

        //执行命令
        this.addCmd.execute();
      //把操作记录到历史记录中
        undoCmds.add(this.addCmd);

        //获取执行命令后的备忘录对象
        Memento m2 = this.addCmd.createMemento();
        //设置到撤销的历史记录中
        this.undoMementos.add(new Memento[]{m1,m2});
    }
    public void substractPressed(){
        //获取对应的备忘录对象，并保存在相应的历史记录中
        Memento m1 = this.substractCmd.createMemento();

        //执行命令
        this.substractCmd.execute();
      //把操作记录到历史记录中
        undoCmds.add(this.substractCmd);

        //获取执行命令后的备忘录对象
        Memento m2 = this.substractCmd.createMemento();
        //设置到撤销的历史记录中
        this.undoMementos.add(new Memento[]{m1,m2});
    }
    public void undoPressed(){
        if(undoCmds.size()>0){
            //取出最后一个命令来撤销
            Command cmd = undoCmds.get(undoCmds.size()-1);
            //获取对应的备忘录对象
            Memento[] ms = undoMementos.get(undoCmds.size()-1);
```

```
            //撤销
            cmd.undo(ms[0]);

            //如果还有恢复的功能，那就把这个命令记录到恢复的历史记录中
            redoCmds.add(cmd);
            //把相应的备忘录对象也添加过去
            redoMementos.add(ms);

            //然后把最后一个命令删除
            undoCmds.remove(cmd);
            //把相应的备忘录对象也删除
            undoMementos.remove(ms);
        }else{
            System.out.println("很抱歉，没有可撤销的命令");
        }
    }
    public void redoPressed(){
        if(redoCmds.size()>0){
            //取出最后一个命令来重做
            Command cmd = redoCmds.get(redoCmds.size()-1);
            //获取对应的备忘录对象
            Memento[] ms = redoMementos.get(redoCmds.size()-1);

            //重做
            cmd.redo(ms[1]);

            //把这个命令记录到可撤销的历史记录中
            undoCmds.add(cmd);
            //把相应的备忘录对象也添加过去
            undoMementos.add(ms);
            //然后把最后一个命令删除
            redoCmds.remove(cmd);
            //把相应的备忘录对象也删除
            redoMementos.remove(ms);
        }else{
            System.out.println("很抱歉，没有可恢复的命令");
        }
    }
}
```

（8）客户端跟以前的实现没有什么变化。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //1：组装命令和接收者
        //创建接收者
        OperationApi operation = new Operation();
        //创建命令
        AddCommand addCmd = new AddCommand(5);
        SubstractCommand substractCmd = new SubstractCommand(3);
        //组装命令和接收者
        addCmd.setOperation(operation);
        substractCmd.setOperation(operation);

        //2：把命令设置到持有者，就是计算器中
        Calculator calculator = new Calculator();
        calculator.setAddCmd(addCmd);
        calculator.setSubstractCmd(substractCmd);

        //3:模拟按下按钮，测试一下
        calculator.addPressed();
        System.out.println("一次加法运算后的结果为："
                              +operation.getResult());
        calculator.substractPressed();
        System.out.println("一次减法运算后的结果为："
                              +operation.getResult());

        //测试撤销
        calculator.undoPressed();
        System.out.println("撤销一次后的结果为："
                              +operation.getResult());
        calculator.undoPressed();
        System.out.println("再撤销一次后的结果为："
                              +operation.getResult());

        //测试恢复
        calculator.redoPressed();
        System.out.println("恢复操作一次后的结果为："
                              +operation.getResult());
        calculator.redoPressed();
        System.out.println("再恢复操作一次后的结果为："
```

```
                         +operation.getResult());
    }
}
```

运行结果示例如下。示例代码如下：

```
一次加法运算后的结果为：5
一次减法运算后的结果为：2
撤销一次后的结果为：5
再撤销一次后的结果为：0
恢复操作一次后的结果为：5
再恢复操作一次后的结果为：2
```

和前面采用补偿式或者反操作式得到的结果是一样的。好好体会一下，对比两种实现方式，看看都是怎么实现的。顺便也体会一下命令模式和备忘录模式是如何结合起来实现功能的。

19.3.5 备忘录模式的优缺点

备忘录模式有以下优点。

- 更好的封装性

  备忘录模式通过使用备忘录对象，来封装原发器对象的内部状态，虽然这个对象是保存在原发器对象的外部，但是由于备忘录对象的窄接口并不提供任何方法。这样有效地保证了对原发器对象内部状态的封装，不把原发器对象的内部实现细节暴露给外部。
- 简化了原发器

  备忘录模式中，备忘录对象被保存到原发器对象之外，让客户来管理他们请求的状态，从而让原发器对象得到简化。
- 窄接口和宽接口

  备忘录模式，通过引入窄接口和宽接口，使得不同的地方，对备忘录对象的访问是不一样的。窄接口保证了只有原发器才可以访问备忘录对象的状态。

备忘录模式的缺点，是可能会导致高开销。

备忘录模式基本的功能，就是对备忘录对象的存储和恢复，它的基本实现方式就是缓存备忘录对象。这样一来，如果需要缓存的数据量很大，或者是特别频繁地创建备忘录对象，开销是很大的。

19.3.6 思考备忘录模式

1. 备忘录模式的本质

> 备忘录模式的本质：**保存和恢复内部状态。**

保存是手段，恢复才是目的，备忘录模式备忘些什么东西呢？

备忘录模式备忘的就是原发器对象的内部状态，这些内部状态是不对外的，只有原发器对象才能够进行操作。

标准的备忘录模式保存数据的手段是，通过内存缓存，广义的备忘录模式实现的时候，可以采用离线存储的方式，把这些数据保存到文件或者数据库等地方。

备忘录模式为何要保存数据呢？目的就是为了在有需要的时候，恢复原发器对象的内部状态。所以恢复是备忘录模式的目的。

根据备忘录模式的本质，从广义上讲，进行数据库存取操作；或者是 Web 应用中的 request、session、servletContext 等的 attribute 数据存取；更进一步，大多数基于缓存功能的数据操作都可以视为广义的备忘录模式。不过广义到这个地步，还提备忘录模式已经没有什么意义了。所以**对于备忘录模式还是多从狭义上来说**。

事实上，对于备忘录模式最主要的一个特点，就是封装状态的备忘录对象，不应该被除了原发器对象之外的对象访问，至于如何存储那都是小事情。因为备忘录模式要解决的主要问题就是：在不破坏对象封装性的前提下，来保存和恢复对象的内部状态，这是一个很主要的判断依据。如果备忘录对象可以让原发器对象以外的对象访问的话，那就算是广义的备忘录模式了，此时提不提备忘录模式已经没有太大的意义了。

2. 何时选用备忘录模式

建议在以下情况中选用备忘录模式。

- 如果必须保存一个对象在某一个时刻的全部或者部分状态，方便在以后需要的时候，可以把该对象恢复到先前的状态，可以使用备忘录模式。使用备忘录对象来封装和保存需要保存的内部状态，然后把备忘录对象保存到管理者对象中，在需要的时候，再从管理者对象中获取备忘录对象，来恢复对象的状态。
- 如果需要保存一个对象的内部状态，但是如果用接口来让其他对象直接得到这些需要保存的状态，将会暴露对象的实现细节并破坏对象的封装性，这时可以使用备忘录模式，把备忘录对象实现成为原发器对象的内部类，而且还是私有的，从而保证只有原发器对象才能访问该备忘录对象。这样既保存了需要保存的状态，又不会暴露原发器对象的内部实现细节。

19.3.7　相关模式

- 备忘录模式和命令模式
  这两个模式可以组合使用。
  命令模式实现中，在实现命令的撤销和重做的时候，可以使用备忘录模式，在命令操作的时候记录下操作前后的状态，然后在命令撤销和重做的时候，直接使用相应的备忘录对象来恢复状态就可以了。
  在这种撤销的执行顺序和重做的执行顺序可控的情况下，备忘录对象还可以采用增量式记录的方式，有效减少缓存的数据量。

研磨

- 备忘录模式和原型模式

  这两个模式可以组合使用。

  在原发器对象创建备忘录对象的时候，如果原发器对象中全部或者大部分的状态都需要保存，一个简洁的方式就是直接克隆一个原发器对象。也就是说，这个时候备忘录对象里面存放的是一个原发器对象的实例，这个在前面已经示例过了，这里就不再赘述。

第 20 章　享元模式（Flyweight）

20.1 场景问题

20.1.1 加入权限控制

考虑这样一个问题，给系统加入权限控制，这基本上是所有的应用系统都有的功能。

对于应用系统而言，一般先要登录系统，才可以使用系统的功能。登录后，用户的每次操作都需要经过权限系统的控制，确保该用户有操作该功能的权限，同时还要控制该用户对数据的访问权限、修改权限等。总之一句话，一个安全的系统，需要对用户的每一次操作都要做权限检测，包括功能和数据，以确保只有获得相应授权的人，才能执行相应的功能，操作相应的数据。

举个例子来说吧，普通人员都有查看本部门人员列表的权限，但是在人员列表中每个人员的薪资数据，普通人员是不可以看到的；而部门经理在查看本部门人员列表的时候，就可以看到每个人员相应的薪资数据。

现在就要来实现为系统加入权限控制的功能，该怎样实现呢？

为了让大家更好地理解后面讲述的知识，先介绍一点权限系统的基础知识。几乎所有的权限系统都分成两个部分，一个是授权部分，一个是验证部分，为了理解它们，首先解释两个基本的名词：安全实体和权限。

- **安全实体**：就是被权限系统检测的对象，比如工资数据。
- **权限**：就是需要被校验的权限对象，比如查看、修改等。

安全实体和权限通常要一起描述才有意义，比如有这样一个描述："现在要检测登录人员对工资数据是否有查看的权限"，"工资数据"这个安全实体和"查看"这个权限一定要一起描述。如果只出现安全实体描述，那就变成这样："现在要检测登录人员对工资数据"，对工资数据干什么呀，没有后半部分，一看就知道不完整；当然只有权限描述也不行，那就变成："现在要检测登录人员是否有查看的权限"，对谁的查看权限啊，也不完整。所以安全实体和权限通常要一起描述。

了解了上面两个名词，下面来看看什么是授权和验证。

- **所谓授权**，是指把对某些安全实体的某些权限分配给某些人员的过程。
- **所谓验证**，是指判断某个人员对某个安全实体是否拥有某个或某些权限的过程。

也就是说，**授权过程即是权限的分配过程，而验证过程则是权限的匹配过程**。在目前应用系统的开发中，多数是利用数据库来存放授权过程产生的数据。也就是说：授权是向数据库中添加数据，或是维护数据的过程，而匹配过程就变成了从数据库中获取相应数据进行匹配的过程了。

为了让问题相对简化一点，就不去考虑权限的另外两个特征，一个是继承性，一个是最近匹配原则。什么意思呢？还是解释一下：

- **权限的继承性**指的是，如果多个安全实体存在包含关系，而某个安全实体没有相应的权限限制，那么它会继承包含它的安全实体的相应权限。

  比如，某个大楼和楼内的房间都是安全实体，很明显大楼这个安全实体会包含楼

内的房间这些安全实体，可以认为大楼是楼内房间的父级实体。现在来考虑一个具体的权限——进入某个房间的权限。如果这个房间没有门，也就是谁都可以进入，相当于这个房间对应的安全实体没有进入房间的权限限制，那么是不是说所有的人都可以进入这个房间呢？当然不是，某人能进入这个房间的前提是。这个人要有权限进入这个大楼。也就是说，此时房间这个安全实体，它本身没有进入权限的限制，但是它会继承父级安全实体的进入权限。

- **权限的最近匹配原则**指的是，如果多个安全实体存在包含关系，而某个安全实体没有相应的权限限制，那么它会向上寻找并匹配相应的权限限制，直到找到一个离这个安全实体最近的拥有相应权限限制的安全实体为止。如果把整个层次结构都寻找完了仍没有匹配到相应权限限制的话，那就说明所有人对这个安全实体都拥有这个相应的权限限制。

  继续上面权限继承性的例子，如果现在这个大楼是坐落在某个机关大院内，这就演变成了，要进入某个房间，首先要有进入大楼的权限，要进入大楼又需要有能进入机关大院的权限。

  所谓最近匹配原则就是，如果某个房间没有门，也就意味着这个房间没有进入的权限限制，那么它就会向上继续寻找并匹配，看看大楼有没有进入的权限限制，如果有就使用这个权限限制，终止寻找；如果没有，继续向上寻找，直到找到一个匹配的为止。如果最后大院也没有进入的权限限制，那就变成所有人都可以进入到这个房间里面来了。

20.1.2　不使用模式的解决方案

1. 看看现在都已经有什么了

系统的授权工作已经完成，授权数据记录在数据库中，具体的数据结构就不去展开了，它记录了人员对安全实体所拥有的权限。假如现在系统中已有如下的授权数据：

```
张三　对　人员列表　拥有　　查看的权限
李四　对　人员列表　拥有　　查看的权限
李四　对　薪资数据　拥有　　查看的权限
李四　对　薪资数据　拥有　　修改的权限
```

2. 思路选择

由于操作人员进行授权操作后，各人员被授予的权限是记录在数据库中的，刚开始由开发人员提出，每次用户操作系统的时候，都直接到数据库中去动态查询，以判断该人员是否拥有相应的权限。但很快就被否决掉了，试想一下，用户操作那么频繁，每次都到数据库中动态查询，这会严重加剧数据库服务器的负担，使系统变慢。

为了加快系统运行的速度，开发小组决定采用一定的缓存。当每个人员登录的时候，就把该人员能操作的权限获取到，存储在内存中。这样每次操作的时候，就直接在内存中进行权限的校验，速度会大大加快，这是**典型的以空间换时间**的做法。

3. 实现示例

（1）首先定义描述授权数据的数据对象。示例代码如下：

```
/**
 * 描述授权数据的数据 model
 */
public class AuthorizationModel {
    /**
     * 人员
     */
    private String user;
    /**
     * 安全实体
     */
    private String securityEntity;
    /**
     * 权限
     */
    private String permit;
    public String getUser() {
        return user;
    }
    public void setUser(String user) {
        this.user = user;
    }
    public String getSecurityEntity() {
        return securityEntity;
    }
    public void setSecurityEntity(String securityEntity) {
        this.securityEntity = securityEntity;
    }
    public String getPermit() {
        return permit;
    }
    public void setPermit(String permit) {
        this.permit = permit;
    }
}
```

相应属性的 getter/setter

（2）为了测试方便，做一个模拟的内存数据库，把授权数据存储在里面，用最简单的字符串存储的方式。示例代码如下：

```
/**
 * 供测试用，在内存中模拟数据库中的值
 */
```

```
public class TestDB {
    /**
     * 用来存放授权数据的值
     */
    public static Collection<String> colDB =
                                    new ArrayList<String>();
    static{
        //通过静态块来填充模拟的数据
        colDB.add("张三,人员列表,查看");
        colDB.add("李四,人员列表,查看");
        colDB.add("李四,薪资数据,查看");
        colDB.add("李四,薪资数据,修改");
        //增加更多的授权数据
        for(int i=0;i<3;i++){
            colDB.add("张三"+i+",人员列表,查看");
        }
    }
}
```

（3）接下来实现登录和权限控制的业务。示例代码如下：

```
/**
 * 安全管理，实现成单例
 */
public class SecurityMgr {
    private static SecurityMgr securityMgr = new SecurityMgr();
    private SecurityMgr(){
    }
    public static SecurityMgr getInstance(){
        return securityMgr;
    }
    /**
     * 在运行期间，用来存放登录人员对应的权限
     * 在 Web 应用中，这些数据通常会存放到 session 中
     */
    private Map<String,Collection<AuthorizationModel>> map =
        new HashMap<String,Collection<AuthorizationModel>>();

    /**
     * 模拟登录的功能
     * @param user 登录的用户
     */
```

```
public void login(String user){
    //登录时就需要把该用户所拥有的权限，从数据库中取出来，放到缓存中去
    Collection<AuthorizationModel> col = queryByUser(user);
    map.put(user, col);
}
/**
 * 判断某用户对某个安全实体是否拥有某种权限
 * @param user 被检测权限的用户
 * @param securityEntity 安全实体
 * @param permit 权限
 * @return true 表示拥有相应权限，false 表示没有相应权限
 */
public boolean hasPermit(String user,String securityEntity
                                        ,String permit){
    Collection<AuthorizationModel> col = map.get(user);
    if(col==null || col.size()==0){
        System.out.println(user+"没有登录或是没有被分配任何权限");
        return false;
    }
    for(AuthorizationModel am : col){
        //输出当前实例，看看是否同一个实例对象
        System.out.println("am=="+am);
        if(am.getSecurityEntity().equals(securityEntity)
                && am.getPermit().equals(permit)){
            return true;
        }
    }
    return false;
}
/**
 * 从数据库中获取某人所拥有的权限
 * @param user 需要获取所拥有的权限的人员
 * @return 某人所拥有的权限
 */
private Collection<AuthorizationModel> queryByUser(
                                        String user){
    Collection<AuthorizationModel> col =
                    new ArrayList<AuthorizationModel>();
    for(String s : TestDB.colDB){
        String ss[] = s.split(",");
```

```
            if(ss[0].equals(user)){
                AuthorizationModel am = new AuthorizationModel();
                am.setUser(ss[0]);
                am.setSecurityEntity(ss[1]);
                am.setPermit(ss[2]);

                col.add(am);
            }
        }
        return col;
    }
}
```

（4）写个客户端来测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //需要先登录，然后再判断是否有权限
        SecurityMgr mgr = SecurityMgr.getInstance();
        mgr.login("张三");
        mgr.login("李四");
        boolean f1 = mgr.hasPermit("张三","薪资数据","查看");
        boolean f2 = mgr.hasPermit("李四","薪资数据","查看");

        System.out.println("f1=="+f1);
        System.out.println("f2=="+f2);
        for(int i=0;i<3;i++){
            mgr.login("张三"+i);
            mgr.hasPermit("张三"+i,"薪资数据","查看");
        }
    }
}
```

运行结果如下：

```
am==cn.javass.dp.flyweight.example1.AuthorizationModel@1eed786
am==cn.javass.dp.flyweight.example1.AuthorizationModel@187aeca
am==cn.javass.dp.flyweight.example1.AuthorizationModel@e48e1b
f1==false
f2==true
am==cn.javass.dp.flyweight.example1.AuthorizationModel@12dacd1
am==cn.javass.dp.flyweight.example1.AuthorizationModel@119298d
am==cn.javass.dp.flyweight.example1.AuthorizationModel@f72617
```

输出结果中的 f1 为 false，表示张三对薪资数据没有查看的权限；而 f2 为 true，表示李四对薪资数据有查看的权限，是正确的，基本完成了功能。

20.1.3 有何问题

看了上面的实现，很简单，而且还考虑了性能的问题，在内存中缓存了每个人相应的权限数据，使得每次判断权限的时候，速度大大加快，实现得挺不错，难道有什么问题吗？

仔细想想，问题就来了，既有缓存这种方式固有的问题，也有我们自己实现上的问题。先说说缓存固有的问题吧。这个不在本次讨论之列，大家了解一下就可以了。

- **缓存时间长度的问题**，就是这些数据应该被缓存多久。如果是 Web 应用，这种跟登录人员相关的权限数据，大多是放在 session 中进行缓存，当 session 超时的时候，就会被清除掉。如果不是 Web 应用呢？就得自己来控制了。另外就算是在 Web 应用中，也不一定非要缓存到 session 超时才清除。总之，控制缓存数据应该被缓存多长时间，是实现高效缓存的一个问题点。
- **缓存数据和真实数据的同步问题**，这里的同步是指数据同步，不是多线程的同步。比如，上面的授权数据是存放在数据库里的，运行的时候缓存到内存中，如果真实的授权数据在运行期间发生了变化，那么缓存中的数据就应该和数据库中的数据同步，以保持一致，否则数据就错了。如何合理地同步数据，也是实现高效缓存的一个问题点。
- **缓存的多线程并发控制**，对于缓存的数据，有些操作从缓存中取值，有些操作向缓存中添加值，有些操作在清除过期的缓存数据，有些操作在进行缓存和真实数据的同步。在一个多线程的环境下，如何合理地对缓存进行并发控制，也是实现高效缓存的一个问题点。

先简单提这么几个。事实上，实现合理、高效的缓存也不是一件很轻松的事情，好在这些问题都不在我们这次的讨论之列。这里的重心还是来讲述模式，而不是缓存实现。

再来看看前面实现上的问题，仔细观察在上面输出结果中框住的部分，这些值是输出对象实例得到的，默认输出的是对象的 hashCode 值，而默认的 hashCode 值可以用来判断是不是同一对象实例。在 Java 中，默认的 equals 方法比较的是内存地址，而 equals 方法和 hashCode 方法的关系是：equals 方法返回 true 的话，那么这两个对象实例的 hashCode 必须相同；而 hashCode 相同，equals 方法并不一定返回 true，也就是说两个对象实例不一定是同一对象实例。换句话说，如果 hashCode 不同的话，肯定不是同一个对象实例。

仔细看看上面的输出结果，框住部分的值是不同的，表明这些对象实例肯定不是同一个对象实例，而是多个对象实例。**这就引出一个问题，就是对象实例数目太多**。为什么这么说呢？看看就描述这么几条数据，数数看有多少个对象实例呢？目前是一条数据就有一个对象实例，这很恐怖。数据库的数据量是很大的，如果有几万条，几十万条，岂不是需要几万个，甚至几十万个对象实例，这样会耗费大量的内存。

另外，这些对象的粒度都很小，都是简单地描述某一个方面的对象，而且很多数据是重复的，在这些大量重复的数据上耗费了很多的内存。比如在前面示例的数据中就会发现有重复的部分，见下面蓝色的部分：

张三	对	人员列表	拥有	查看的权限
李四	对	人员列表	拥有	查看的权限
李四	对	薪资数据	拥有	查看的权限
李四	对	薪资数据	拥有	修改的权限

前面讲过，对于安全实体和权限一般要联合描述，因此对于“人员列表 这个安全实体 的 查看权限 限制”，就算是授权给不同的人员，这个描述是一样的。假设在某极端情况下，要把“人员列表 这个安全实体 的 查看权限 限制”授权给一万个人，那么数据库中将会有一万条记录，按照前面的实现方式，会有一万个对象实例，而这些实例中，有大部分的数据是重复的，而且会重复一万次，你觉得这是不是个很大的问题呢？

把上面的问题描述出来就是：在系统当中，存在大量的细粒度对象，而且存在大量的重复数据，严重耗费内存，如何解决呢？

20.2 解决方案

20.2.1 使用享元模式来解决问题

用来解决上述问题的一个合理的解决方案就是享元模式。那么什么是享元模式呢？

1. 享元模式的定义

> 运用共享技术有效地支持大量细粒度的对象。

2. 应用享元模式来解决的思路

仔细观察和分析上面的授权信息，会发现有一些数据是重复出现的，比如：人员列表、薪资数据、查看、修改等。至于人员相关的数据，考虑到每个描述授权的对象都是和某个人员相关的，所以存放的时候，会把相同人员的授权信息组织在一起，就不去考虑人员数据的重复性了。

现在造成内存浪费的主要原因：就是细粒度对象太多，而且有大量重复的数据。如果能够有效地减少对象的数量，减少重复的数据，那么就能够节省不少内存。一个基本的思路就是缓存这些包含着重复数据的对象，让这些对象只出现一次，也就只耗费一份内存了。

> 注意　**但是请注意，并不是所有的对象都适合缓存，因为缓存的是对象的实例，实例里面存放的主要是对象属性的值。因此，如果被缓存的对象的属性值经常变动，那就不适合缓存了，因为真实对象的属性值变化了，那么缓存中的对象也必须要跟着变化，否则缓存中的数据就跟真实对象的数据不同步，可以说是错误的数据了。**

因此，需要分离出被缓存对象实例中，哪些数据是不变且重复出现的，哪些数据是经常变化的，真正应该被缓存的数据是那些不变且重复出现的数据，把它们称为对象的内部状态，而那些变化的数据就不缓存了，把它们称为对象的外部状态。

这样在实现的时候，把内部状态分离出来共享，称之为享元，通过共享享元对象来减少对内存的占用。把外部状态分离出来，放到外部，让应用在使用的时候进行维护，并在需要的时候传递给享元对象使用。为了控制对内部状态的共享，并且让外部能简单地使用共享数据，提供一个工厂来管理享元，把它称为享元工厂。

20.2.2 享元模式的结构和说明

享元模式的结构如图 20.1 所示。

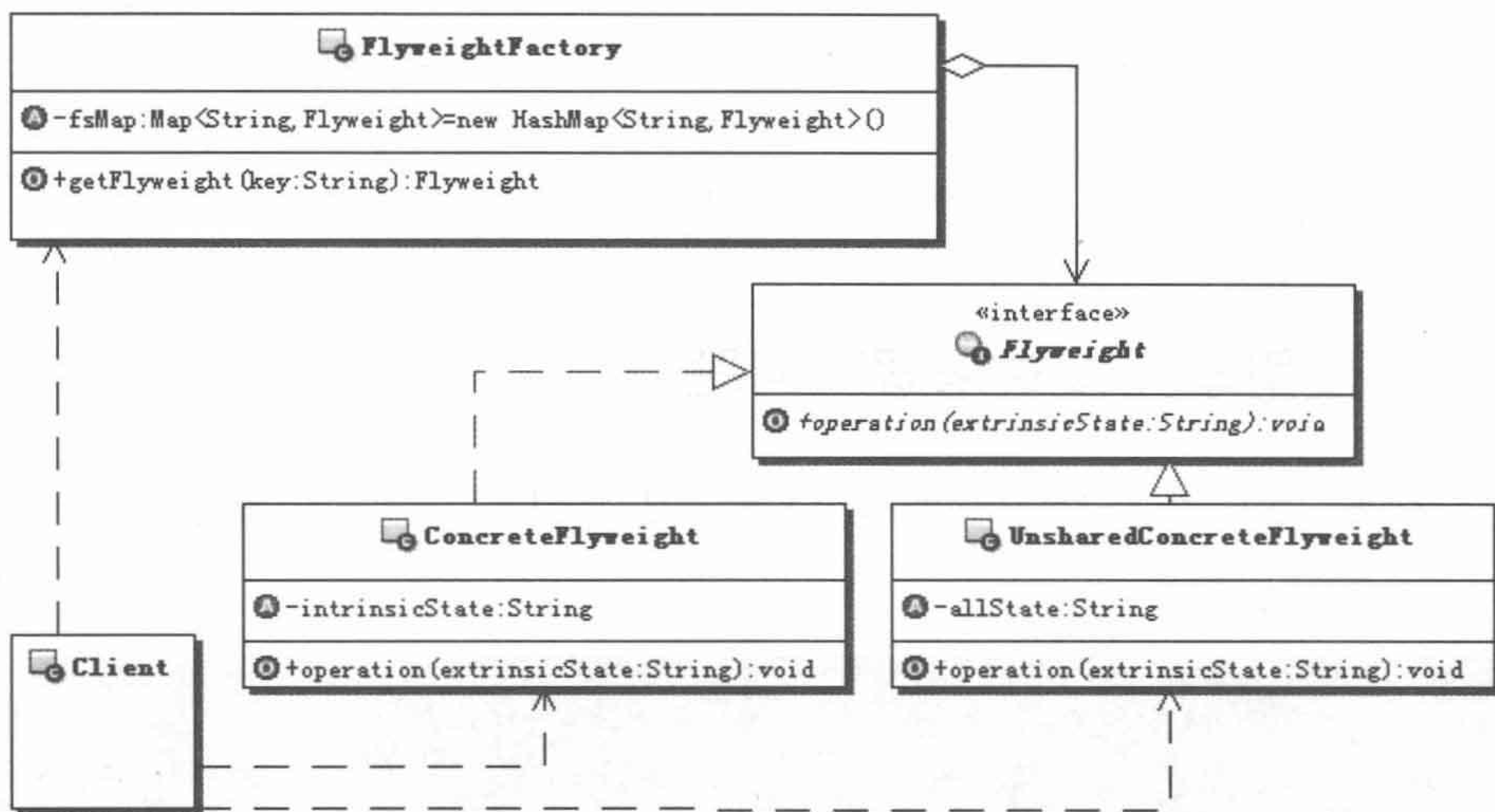


图 20.1　享元模式的结构图

- Flyweight：享元接口，通过这个接口 Flyweight 可以接受并作用于外部状态。通过这个接口传入外部的状态，在享元对象的方法处理中可能会使用这些外部的数据。
- ConcreteFlyweight：具体的享元实现对象，必须是可共享的，需要封装 Flyweight 的内部状态。
- UnsharedConcreteFlyweight：非共享的享元实现对象，并不是所有的 Flyweight 实现对象都需要共享。非共享的享元实现对象通常是对共享享元对象的组合对象。
- FlyweightFactory：享元工厂，主要用来创建并管理共享的享元对象，并对外提供访问共享享元的接口。
- Client：享元客户端，主要的工作是维持一个对 Flyweight 的引用，计算或存储享元对象的外部状态，当然这里可以访问共享和不共享的 Flyweight 对象。

20.2.3 享元模式示例代码

（1）先看看享元的接口定义。通过这个接口 Flyweight 可以接受并作用于外部状态。示例代码如下：

```
/***
 * 享元接口，通过这个接口享元可以接受并作用于外部状态
 */
public interface Flyweight {
    /**
     * 示例操作，传入外部状态
     * @param extrinsicState 示例参数，外部状态
     */
    public void operation(String extrinsicState);
}
```

（2）接下来看看具体的享元接口的实现。

先看看共享享元的实现。封装 Flyweight 的内部状态，当然也可以提供功能方法。示例代码如下：

```
/**
 * 享元对象
 */
public class ConcreteFlyweight implements Flyweight{
    /**
     * 示例，描述内部状态
     */
    private String intrinsicState;
    /**
     * 构造方法，传入享元对象的内部状态的数据
     * @param state 享元对象的内部状态的数据
     */
    public ConcreteFlyweight(String state){
        this.intrinsicState = state;
    }

    public void operation(String extrinsicState) {
        //具体的功能处理，可能会用到享元内部、外部的状态
    }
}
```

再来看看不需要共享的享元对象的实现。并不是所有的 Flyweight 对象都需要共享，Flyweight 接口使共享成为可能，但并不强制共享。示例代码如下：

```
/**
 * 不需要共享的 flyweight 对象，
 * 通常是将被共享的享元对象作为子节点组合出来的对象
 */
```

```
public class UnsharedConcreteFlyweight implements Flyweight{
    /**
     * 示例，描述对象的状态
     */
    private String allState;

    public void operation(String extrinsicState) {
        // 具体的功能处理
    }
}
```

（3）在享元模式中，客户端不能直接创建共享的享元对象实例，必须通过享元工厂来创建。下面来看看享元工厂的实现。示例代码如下：

```
/**
 * 享元工厂
 */
public class FlyweightFactory {
    /**
     * 缓存多个Flyweight对象，这里只是示意一下
     */
    private Map<String,Flyweight> fsMap =
                                  new HashMap<String,Flyweight>();
    /**
     * 获取key对应的享元对象
     * @param key 获取享元对象的key，只是示意
     * @return key 对应的享元对象
     */
    public Flyweight getFlyweight(String key) {
        //这个方法中基本的实现步骤如下：
        //1：先从缓存中查找，是否存在key对应的Flyweight对象
        Flyweight f = fsMap.get(key);

        //2：如果存在，就返回相对应的Flyweight对象
        if(f==null){
            //3：如果不存在
            //3.1：创建一个新的Flyweight对象
            f = new ConcreteFlyweight(key);
            //3.2：把这个新的Flyweight对象添加到缓存中
            fsMap.put(key,f);
            //3.3：然后返回这个新的Flyweight对象
        }
```

```
            return f;
        }
}
```

（4）最后来看看客户端的实现。客户端通常会维持一个对 Flyweight 的引用，计算或存储一个或多个 Flyweight 的外部状态。示例代码如下：

```
/**
 * Client 对象，通常会维持一个对 Flyweight 的引用，
 * 计算或存储一个或多个 Flyweight 的外部状态
 */
public class Client {
    //具体的功能处理
}
```

20.2.4　使用享元模式重写示例

再次分析上面的授权信息。实际上重复出现的数据主要是对安全实体和权限的描述，又考虑到安全实体和权限的描述一般是不分开的，那么找出这些重复的描述，比如，人员列表的查看权限。而且这些重复的数据是可以重用的，比如给它们配上不同的人员，就可以组合成为不同的授权描述，如图 20.2 所示。



图 20.2　授权描述示意图

图 20.2 就可以描述如下的信息：

```
张三  对  人员列表  拥有  查看的权限
李四  对  人员列表  拥有  查看的权限
王五  对  人员列表  拥有  查看的权限
```

很明显，可以把安全实体和权限的描述定义成为享元，而和它们结合的人员数据，就可以作为享元的外部数据。为了演示简单，就把安全实体对象和权限对象简化成了字符串，描述一下它们的名字。

（1）按照享元模式，也为了系统的扩展性和灵活性，给享元定义一个接口，外部使用享元还是面向接口来编程。示例代码如下：

```
/***
 * 描述授权数据的享元接口
 */
```

```
public interface Flyweight {
    /**
     * 判断传入的安全实体和权限，是否和享元对象内部状态匹配
     * @param securityEntity 安全实体
     * @param permit 权限
     * @return true 表示匹配，false 表示不匹配
     */
    public boolean match(String securityEntity,String permit);
}
```

（2）定义了享元接口，该来实现享元对象了，这个对象需要封装授权数据中重复出现部分的数据。示例代码如下：

```
/**
 * 封装授权数据中重复出现部分的享元对象
 */
public class AuthorizationFlyweight implements Flyweight{
    /**
     * 内部状态，安全实体
     */
    private String securityEntity;
    /**
     * 内部状态，权限
     */
    private String permit;
    /**
     * 构造方法，传入状态数据
     * @param state 状态数据，包含安全实体和权限的数据，用","分隔
     */
    public AuthorizationFlyweight(String state){
        String ss[] = state.split(",");
        securityEntity = ss[0];
        permit = ss[1];
    }

    public String getSecurityEntity() {
        return securityEntity;
    }
    public String getPermit() {
        return permit;
    }
```

根据需要，可以提供外部访问内部数据的 getter 方法，但是不会提供 setter 方法，也就是这些数据不会让外部来修改

```
    public boolean match(String securityEntity, String permit) {
        if(this.securityEntity.equals(securityEntity)
                && this.permit.equals(permit)){
            return true;
        }
        return false;
    }
}
```

（3）定义好了享元，来看看如何管理这些享元。提供享元工厂来负责享元对象的共享管理和对外提供访问享元的接口。

享元工厂一般不需要很多个，实现成为单例即可。享元工厂负责享元对象的创建和管理，基本的思路就是在享元工厂中缓存享元对象。在 Java 中最常用的缓存实现方式，就是定义一个 Map 来存放缓存的数据，而享元工厂对外提供的访问享元的接口，基本上就是根据 key 值到缓存的 Map 中获取相应的数据，这样只要有了共享，同一份数据就可以重复使用了。示例代码如下：

```
/**
 * 享元工厂，通常实现成为单例
 */
public class FlyweightFactory {
    private static FlyweightFactory factory =
                                        new FlyweightFactory();
    private FlyweightFactory(){
    }
    public static FlyweightFactory getInstance(){
        return factory;
    }
    /**
     * 缓存多个 Flyweight 对象
     */
    private Map<String,Flyweight> fsMap =
                                new HashMap<String,Flyweight>();
    /**
     * 获取 key 对应的享元对象
     * @param key 获取享元对象的 key
     * @return key 对应的享元对象
     */
    public Flyweight getFlyweight(String key) {
        Flyweight f = fsMap.get(key);
        if(f==null){
```

外部通过这个方法来获取享元对象

```
            f = new AuthorizationFlyweight(key);
            fsMap.put(key,f);
        }
        return f;
    }
}
```

（4）使用享元对象。

实现完享元工厂，该来看看如何使用享元对象了。按照前面的实现，需要一个对象来提供安全管理的业务功能，就是前面的那个 SecurityMgr 类，这个类现在在享元模式中，就充当了 Client 的角色。注意这个 Client 角色和我们平时说的测试客户端是两个概念，这个 Client 角色是使用享元的对象。

SecurityMgr 的实现方式基本上模仿前面的实现，也会有相应的改变，变化大致如下。

- 缓存的每个人员的权限数据，类型变成了 Flyweight 的。
- 在原来 queryByUser 方法中，通过 new 来创建授权对象的地方修改成了通过享元工厂来获取享元对象，这是使用享元模式最重要的一点改变，也就是不是直接去创建对象实例，而是通过享元工厂来获取享元对象实例。

示例代码如下：

```
/**
 * 安全管理，实现成单例
 */
public class SecurityMgr {
    private static SecurityMgr securityMgr = new SecurityMgr();
    private SecurityMgr(){
    }
    public static SecurityMgr getInstance(){
        return securityMgr;
    }
    /**
     * 在运行期间，用来存放登录人员对应的权限
     * 在 Web 应用中，这些数据通常会存放到 session 中
     */
    private Map<String,Collection<Flyweight>> map =
        new HashMap<String,Collection<Flyweight>>();
    /**
     * 模拟登录的功能
     * @param user 登录的用户
     */
    public void login(String user){
        //登录时就需要把该用户所拥有的权限，从数据库中取出来，放到缓存中去
```

```
        Collection<Flyweight> col = queryByUser(user);
        map.put(user, col);
    }
    /**
     * 判断某用户对某个安全实体是否拥有某种权限
     * @param user 被检测权限的用户
     * @param securityEntity 安全实体
     * @param permit 权限
     * @return true 表示拥有相应权限，false 表示没有相应权限
     */
    public boolean hasPermit(String user,String securityEntity
                                            ,String permit){
        Collection<Flyweight> col = map.get(user);
        if(col==null || col.size()==0){
            System.out.println(user+"没有登录或是没有被分配任何权限");
            return false;
        }
        for(Flyweight fm : col){
            //输出当前实例，看看是否同一个实例对象
            System.out.println("fm=="+fm);
            if(fm.match(securityEntity, permit)){
                return true;
            }
        }
        return false;
    }
    /**
     * 从数据库中获取某人所拥有的权限
     * @param user 需要获取所拥有的权限的人员
     * @return 某人所拥有的权限
     */
    private Collection<Flyweight> queryByUser(String user){
        Collection<Flyweight> col = new ArrayList<Flyweight>();
        for(String s : TestDB.colDB){
            String ss[] = s.split(",");
            if(ss[0].equals(user)){
                Flyweight fm = FlyweightFactory.getInstance()
                                    .getFlyweight(ss[1]+","+ss[2]);

                col.add(fm);
```

```
                }
            }
            return col;
        }
}
```

（5）所用到的 TestDB 没有任何变化，这里不再赘述。

（6）客户端测试代码也没有任何变化，也不再赘述。

运行测试一下，看看效果。主要是看看是不是能有效地减少那些重复数据对象的数量。运行结果如下：

```
fm==cn.javass.dp.flyweight.example3.AuthorizationFlyweight@e48e1b
fm==cn.javass.dp.flyweight.example3.AuthorizationFlyweight@e48e1b
fm==cn.javass.dp.flyweight.example3.AuthorizationFlyweight@12dacd1
f1==false
f2==true
fm==cn.javass.dp.flyweight.example3.AuthorizationFlyweight@e48e1b
fm==cn.javass.dp.flyweight.example3.AuthorizationFlyweight@e48e1b
fm==cn.javass.dp.flyweight.example3.AuthorizationFlyweight@e48e1b
```

仔细观察结果中蓝色的部分，会发现六条数据中，有五条的 hashCode 是同一个值，根据我们的实现，可以断定这是同一个对象。也就是说，现在只有两个对象实例，而前面的实现中有六个对象实例。

如同示例的那样，对于封装安全实体和权限的这些细粒度对象，既是授权分配的单元对象，也是权限检测的单元对象。可能有很多人对某个安全实体拥有某个权限，如果为每个人都重新创建一个对象来描述对应的安全实体和权限，那样就太浪费内存空间了。

通过共享封装了安全实体和权限的对象，无论多少人拥有这个权限，实际的对象实例都是只有一个，这样既减少了对象的数目，又节省了宝贵的内存空间，从而解决了前面提出的问题。

20.3 模式讲解

20.3.1 认识享元模式

1. 变与不变

享元模式设计的重点就在于分离变与不变。把一个对象的状态分成内部状态和外部状态，内部状态是不变的，外部状态是可变的。然后通过共享不变的部分，达到减少对象数量并节约内存的目的。在享元对象需要的时候，可以从外部传入外部状态给共享的对象，共享对象会在功能处理的时候，使用自己内部的状态和这些外部的状态。

事实上，分离变与不变是软件设计上最基本的方式之一，比如预留接口，为什么在

这个地方要预留接口，一个常见的原因就是这里存在变化，可能在今后需要扩展或者是改变已有的实现，因此预留接口作为“可插入性的保证”。

2. 共享与不共享

在享元模式中，享元对象又有共享与不共享之分，这种情况通常出现在和组合模式合用的情况，通常共享的是叶子对象，一般不共享的部分是由共享部分组合而成的，由于所有细粒度的叶子对象都已经缓存了，那么缓存组合对象就没有什么意义了。这在后面将给大家一个示例。

3. 内部状态和外部状态

享元模式的内部状态，通常指的是包含在享元对象内部的、对象本身的状态，是独立于使用享元的场景的信息，一般创建后就不再变化的状态，因此可以共享。

外部状态指的是享元对象之外的状态，取决于使用享元的场景，会根据使用场景而变化，因此不可共享。如果享元对象需要这些外部状态的话，可以从外部传递到享元对象中，比如通过方法的参数来传递。

也就是说享元模式真正缓存和共享的数据是享元的内部状态，而外部状态是不应该被缓存共享的。

还有一点，内部状态和外部状态是独立的，外部状态的变化不应该影响到内部状态。

4. 实例池

在享元模式中，为了创建和管理共享的享元部分，引入了享元工厂。享元工厂中一般都包含有享元对象的实例池，享元对象就是缓存在这个实例池中的。

简单介绍一点实例池的知识。**所谓实例池，指的是缓存和管理对象实例的程序，通常实例池会提供对象实例的运行环境，并控制对象实例的生命周期。**

> 延伸　**工业级的实例池在实现上有两个最基本的难点，一个是动态控制实例数量，另一个是动态分配实例来提供给外部使用。这些都是需要算法来做保证的。**

假如实例池中已有了 3 个实例，但是客户端请求非常多，有些忙不过来，那么实例池的管理程序就应该判断，到底几个实例才能满足现在的客户需求，理想状况是刚刚好，就是既能够满足应用的需要，又不会造成对象实例的浪费。假如经过判断 5 个实例正好，那么实例池的管理程序就应该能动态地创建 2 个新的实例。

这样运行了一段时间，客户端的请求减少了，这个时候实例池的管理程序又该动态地判断，究竟几个实例是最好的，多了明显浪费资源。假如经过判断只需要 1 个实例就可以了，那么实例池的管理程序应该销毁掉多余的 4 个实例，以释放资源。这就是动态控制实例数量。

对于动态分配实例，也说明一下。假如实例池中有 3 个实例，这个时候来了一个新的请求，到底调度哪一个实例去执行客户的请求呢？如果有空闲实例，那就调度空闲实例去执行客户的请求，如果没有空闲实例呢，是新建一个实例，还是等待运行中的实例，等它运行完了就来处理这个请求呢？具体如何调度，也是需要算法来保障的。

回到享元模式中来，享元工厂中的实例池并没有这么复杂，因为共享的享元对象基

本上都是一个实例，一般不会出现同一个享元对象有多个实例的情况。这样就不用去考虑动态创建和销毁享元对象实例的功能；另外因为只有一个实例，也就不存在动态调度的麻烦，反正就是它了。

这也主要是因为享元对象封装的多半是对象的内部状态，这些状态通常是不变的，有一个实例就够了，不需要动态控制生命周期，也不需要动态调度，它只需要做一个缓存而已，没有上升到真正的实例池的高度。

5. 享元模式的调用顺序示意图

享元模式的使用上，有两种情况，一种是没有“不需要共享”的享元对象，就如同前面的示例那样，只有共享享元对象的情况；还有一种是既有共享享元对象，又有不需要共享的享元对象的情况，这种情况后面再示例。

下面看看只有共享享元对象的情况下，享元模式的调用顺序，如图 20.3 所示。

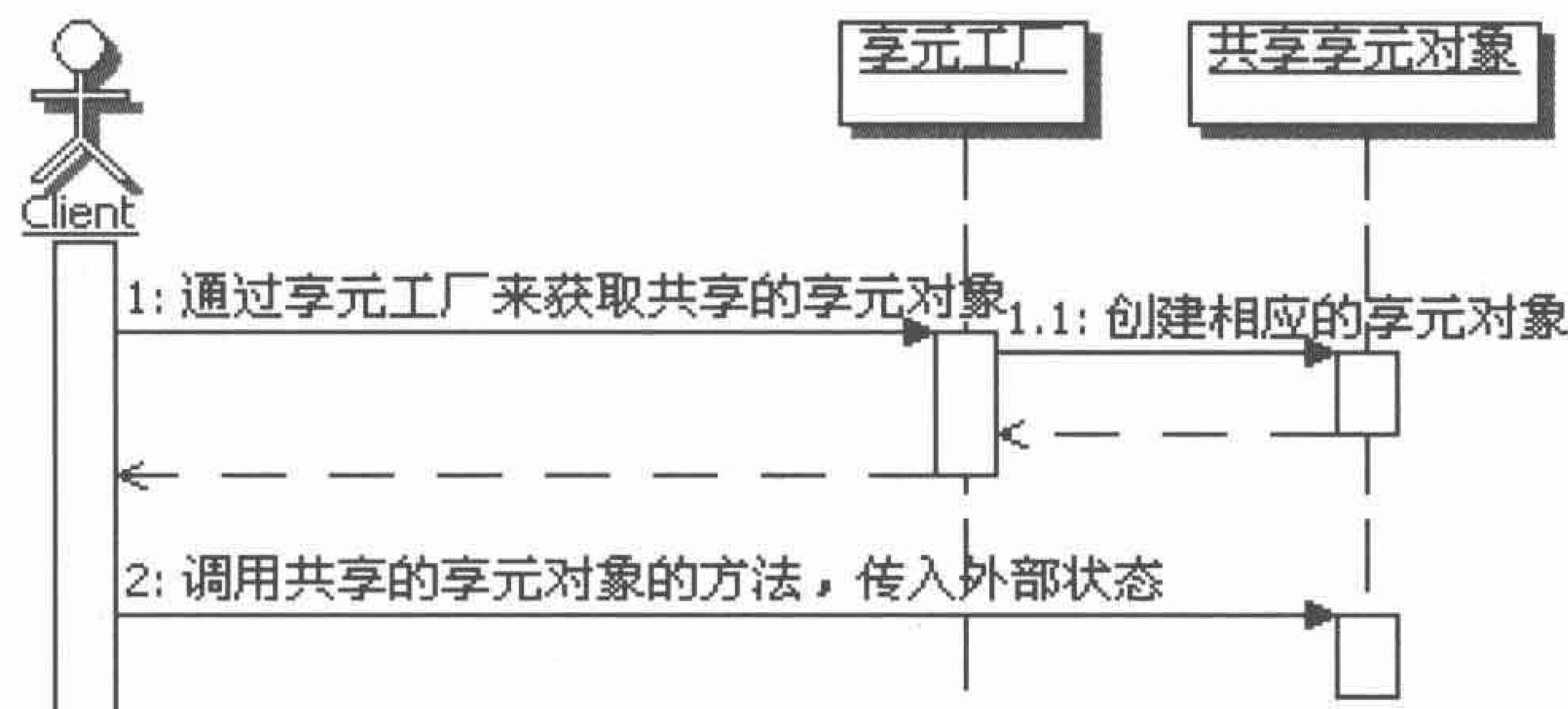


图 20.3　只有共享享元对象的情况下享元模式的调用顺序示意图

6. 谁来初始化共享对象

在享元模式中，通常是在第一次向享元工厂请求获取共享对象的时候，进行共享对象的初始化，而且多半都是在享元工厂内部实现，不会从外部传入共享对象。当然可以从外部传入一些创建共享对象需要的值，享元工厂可以按照这些值去初始化需要共享的对象，然后把创建好的共享对象的实例放入享元工厂内部的缓存中，以后再请求这个共享对象的时候就不用再创建了。

20.3.2　不需要共享的享元实现

可能有些朋友看到这个标题会很疑惑，享元不就是要共享的对象吗？不共享，叫什么享元啊？

确实有不需要共享的享元实现，这种情况多出现在组合结构中，对于使用已经缓存的享元组合出来的对象，就没有必要再缓存了。也就是把已经缓存的享元当做叶子结点，组合出来的组合对象就不需要再被缓存了。也把这种享元称为复合享元。

比如上面的权限描述，如果出现组合权限描述，在这个组合对象中包含很多个共享的权限描述，那么这个组合对象就不用缓存了，该组合对象的存在只是为了在授权的时候更加方便。

具体点说吧，比如要给某人分配“薪资数据”这个安全实体的“修改”权限，那么一定会把“薪资数据”的“查看权限”也分配给这个人。如果按照前面的做法，需要分

配两个对象，为了方便，干脆把这两个描述组合起来，打包成一个对象，命名为“操作薪资数据”，那么分配权限的时候，可以这样描述：

```
把　“操作薪资数据”　分配给　张三
```

这句话的意思就相当于：

```
把　“薪资数据”　的　“查看”权限　分配给　张三
把　“薪资数据”　的　“修改”权限　分配给　张三
```

这样一来，“操作薪资数据”就相当于是一个不需要共享的享元，它实际由享元“薪资数据 的 查看 权限”和享元“薪资数据 的 修改 权限”这两个享元组合而成，因此“操作薪资数据”本身也就不需要再共享了。

这样分配权限的时候就会简单一点。

但是这种组合对象在权限系统中一般不用于验证，也就是说验证的时候还是一个一个进行判断，因为在存储授权信息的时候是一条一条存储的。但也不排除有些时候始终要检查多个权限，干脆把这些权限打包，然后直接验证是否有这个组合权限，只是这种情况应用得比较少而已。

还是用示例来说明吧。在上面已经实现的系统中添加不需要共享的享元实现。此时系统结构如图 20.4 所示。

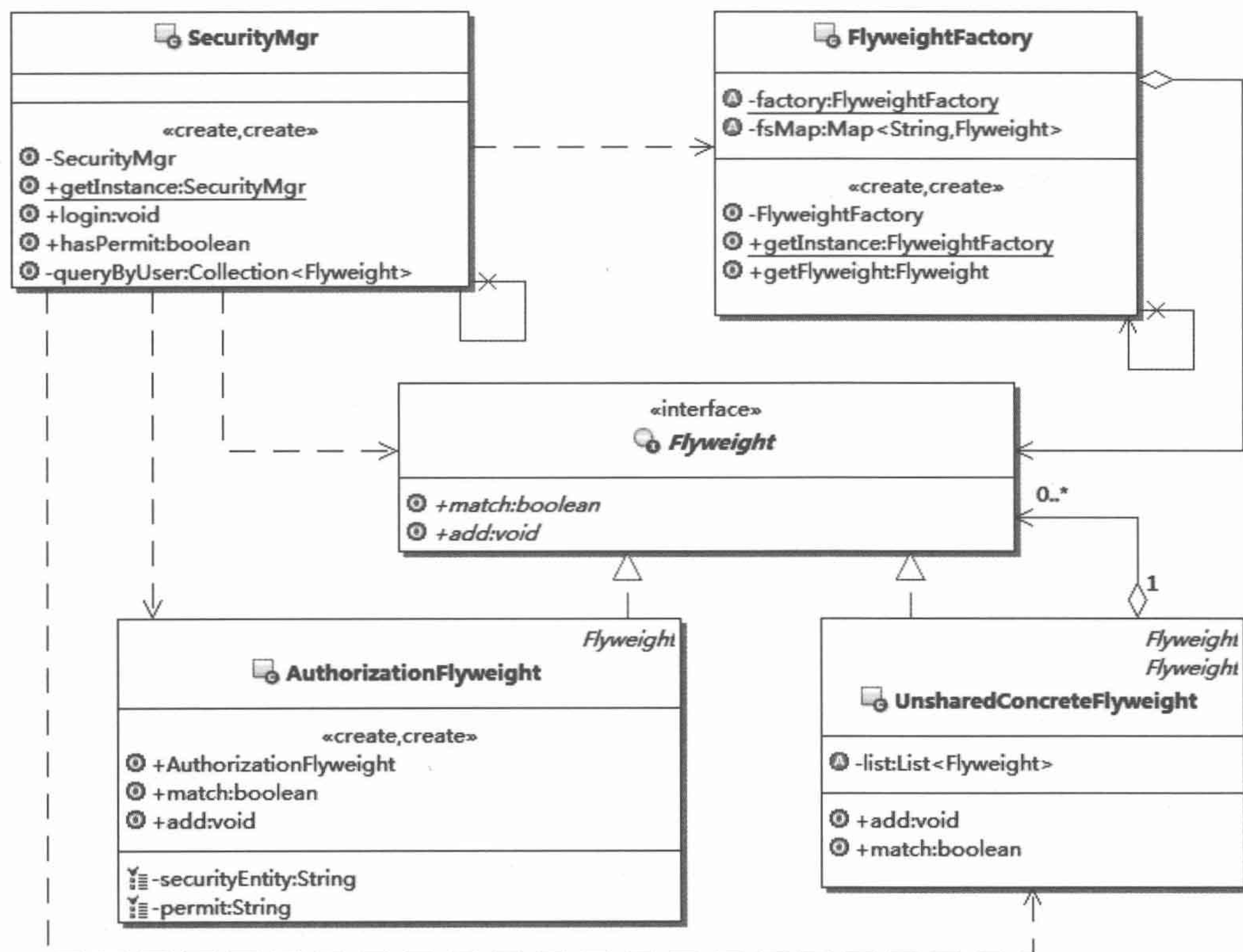


图 20.4　不需要共享享元的示例机构示意图

（1）首先要在享元接口上添加对组合对象的操作，主要是添加向组合对象中加入子对象的方法。示例代码如下：

```
/***
 * 描述授权数据的享元接口
 */
public interface Flyweight {
    /**
     * 判断传入的安全实体和权限，是否和享元对象的内部状态匹配
     * @param securityEntity 安全实体
     * @param permit 权限
     * @return true 表示匹配，false 表示不匹配
     */
    public boolean match(String securityEntity,String permit);
    /**
     * 为 Flyweight 添加子 Flyweight 对象
     * @param f 被添加的子 Flyweight 对象
     */
    public void add(Flyweight f);
}
```

（2）享元接口改变了，那么原来共享的享元对象也需要实现这个方法，这个方法主要是针对组合对象的，因此在叶子对象中抛出不支持的例外就可以了。示例代码如下：

```
/**
 * 封装授权数据中重复出现部分的享元对象
 */
public class AuthorizationFlyweight implements Flyweight{

    其他部分没有任何变化，为了篇幅关系，省略了，
    下面的方法是新加入的

    public void add(Flyweight f) {
        throw new UnsupportedOperationException(
                                        "对象不支持这个功能");
    }
}
```

（3）接下来实现新的不需要共享的享元对象，其实就是组合共享享元对象的对象，这个组合对象中，需要保存所有的子对象，另外它在实现 match 方法的时候，是通过递归的方式，在整个组合结构中进行匹配。示例代码如下：

```
/**
 * 不需要共享的享元对象的实现，也是组合模式中的组合对象
 */
public class UnsharedConcreteFlyweight implements Flyweight{
    /**
     * 记录每个组合对象所包含的子组件
     */
    private List<Flyweight> list = new ArrayList<Flyweight>();

    public void add(Flyweight f) {
        list.add(f);
    }

    public boolean match(String securityEntity, String permit) {
        for(Flyweight f : list){
            //递归调用
            if(f.match(securityEntity, permit)){
                return true;
            }
        }
        return false;
    }
}
```

（4）在继续实现之前，先来准备测试数据，也就是 TestDB，需要有一些改变。

首先是授权数据要区分是单条的授权还是组合的授权，这个在每条授权数据后面添加一个标识来描述。

然后增加一个描述组合数据的记录，使用一个 Map 来存放。

具体的示例代码如下：

```
/**
 * 供测试用，在内存中模拟数据库中的值
 */
public class TestDB {
    /**
     * 用来存放单独授权数据的值
     */
    public static Collection<String> colDB =
                                       new ArrayList<String>();
    /**
     * 用来存放组合授权数据的值
```

```
     * key为组合数据的id，value为该组合包含的多条授权数据的值
     */
    public static Map<String,String[]> mapDB =
                                    new HashMap<String,String[]>();
    static{
        //通过静态块来填充模拟的数据，增加一个标识来表明是否组合授权数据
        colDB.add("张三,人员列表,查看,1");
        colDB.add("李四,人员列表,查看,1");
        colDB.add("李四,操作薪资数据,,2");

        mapDB.put("操作薪资数据",
                new String[]{"薪资数据,查看","薪资数据,修改"});

        //增加更多的授权数据
        for(int i=0;i<3;i++){
            colDB.add("张三"+i+",人员列表,查看,1");
        }
    }
}
```

（5）享元工厂不需要变化，这里就不再赘述。

（6）接下来该实现安全管理的类了，这个类相当于享元模式的 Client 角色。这次在这个类中，不仅会使用共享的享元对象，它还会使用不需要共享的享元对象。

主要的变化集中在 queryByUser 方法中，。原本只是通过享元工厂来获取共享的享元对象即可，但这次还需要在这里创建不需要共享的享元对象。示例代码如下：

```
public class SecurityMgr {
    private static SecurityMgr securityMgr = new SecurityMgr();
    private SecurityMgr(){
    }
    public static SecurityMgr getInstance(){
        return securityMgr;
    }
    /**
     * 在运行期间，用来存放登录人员对应的权限
     * 在Web应用中，这些数据通常会存放到session中
     */
    private Map<String,Collection<Flyweight>> map =
        new HashMap<String,Collection<Flyweight>>();
    /**
     * 模拟登录的功能
```

```
 * @param user 登录的用户
 */
public void login(String user){
    //登录时就需要把该用户所拥有的权限，从数据库中取出来，放到缓存中去
    Collection<Flyweight> col = queryByUser(user);
    map.put(user, col);
}
/**
 * 判断某用户对某个安全实体是否拥有某种权限
 * @param user 被检测权限的用户
 * @param securityEntity 安全实体
 * @param permit 权限
 * @return true 表示拥有相应权限，false 表示没有相应权限
 */
public boolean hasPermit(String user,String securityEntity
                                          ,String permit){
    Collection<Flyweight> col = map.get(user);
    System.out.println("现在测试"+securityEntity+"的"+permit
                                  +"权限，map.size="+map.size());
    if(col==null || col.size()==0){
        System.out.println(user+"没有登录或是没有被分配任何权限");
        return false;
    }
    for(Flyweight fm : col){
        //输出当前实例，看看是否同一个实例对象
        System.out.println("fm=="+fm);
        if(fm.match(securityEntity, permit)){
            return true;
        }
    }
    return false;
}
/**
 * 从数据库中获取某人所拥有的权限
 * @param user 需要获取所拥有的权限的人员
 * @return 某人所拥有的权限
 */
private Collection<Flyweight> queryByUser(String user){
    Collection<Flyweight> col = new ArrayList<Flyweight>();
    for(String s : TestDB.colDB){
```

```java
                String ss[] = s.split(",");
                if(ss[0].equals(user)){
                    Flyweight fm = null;
                    if(ss[3].equals("2")){
                        //表示是组合
                        fm = new UnsharedConcreteFlyweight();
                        //获取需要组合的数据
                        String tempSs[] = TestDB.mapDB.get(ss[1]);
                        for(String tempS : tempSs){
                            Flyweight tempFm = FlyweightFactory
                                    .getInstance().getFlyweight(tempS);
                            //把这个对象加入到组合对象中
                            fm.add(tempFm);
                        }
                    }else{
                        fm = FlyweightFactory.getInstance()
                                .getFlyweight(ss[1]+","+ss[2]);
                    }

                    col.add(fm);
                }
            }
            return col;
        }
}
```

（7）客户端测试没有太大的变化，增加一条测试“李四对薪资数据的修改权限”。示例代码如下：

```java
public class Client {
    public static void main(String[] args) throws Exception{
        //需要先登录，然后再判断是否有权限
        SecurityMgr mgr = SecurityMgr.getInstance();
        mgr.login("张三");
        mgr.login("李四");
        boolean f1 = mgr.hasPermit("张三","薪资数据","查看");
        boolean f2 = mgr.hasPermit("李四","薪资数据","查看");
        boolean f3 = mgr.hasPermit("李四","薪资数据","修改");

        System.out.println("f1=="+f1);
        System.out.println("f2=="+f2);
        System.out.println("f3=="+f3);
```

```
        for(int i=0;i<3;i++){
            mgr.login("张三"+i);
            mgr.hasPermit("张三"+i,"薪资数据","查看");
        }
    }
}
```

可以运行测试一下，看看效果。结果示例如下：

```
现在测试薪资数据的查看权限，map.size=2
fm==cn.javass.dp.flyweight.example4.AuthorizationFlyweight@12dacd1
现在测试薪资数据的查看权限，map.size=2
fm==cn.javass.dp.flyweight.example4.AuthorizationFlyweight@12dacd1
fm==cn.javass.dp.flyweight.example4.UnsharedConcreteFlyweight@1ad0
86a
现在测试薪资数据的修改权限，map.size=2
fm==cn.javass.dp.flyweight.example4.AuthorizationFlyweight@12dacd1
fm==cn.javass.dp.flyweight.example4.UnsharedConcreteFlyweight@1ad0
86a
f1==false
f2==true
f3==true
现在测试薪资数据的查看权限，map.size=3
fm==cn.javass.dp.flyweight.example4.AuthorizationFlyweight@12dacd1
现在测试薪资数据的查看权限，map.size=4
fm==cn.javass.dp.flyweight.example4.AuthorizationFlyweight@12dacd1
现在测试薪资数据的查看权限，map.size=5
fm==cn.javass.dp.flyweight.example4.AuthorizationFlyweight@12dacd1
```

20.3.3　对享元对象的管理

虽然享元模式对于共享的享元对象实例的管理要求没有实例池对实例管理的要求那么高，但是也还是有很多自身的特点功能，比如，引用计数、垃圾清除等。**所谓垃圾，就是在缓存中存在，但是不再需要被使用的缓存中的对象。**

所谓引用计数，就是享元工厂能够记录每个享元被使用的次数；而垃圾清除，则是大多数缓存管理都有的功能，缓存不能只往里面放数据，在不需要这些数据的时候，应该把这些数据从缓存中清除，释放相应的内存空间，以节约资源。

在前面的示例中，共享的享元对象是很多人共享的，基本上可以一直存在于系统中，不用清除。但是垃圾清除是享元对象管理的一个常见的功能。继续通过示例给大家讲一下，看看如何实现这些常见的功能。

1. 实现引用计数的基本思路

要实现引用计数，就在享元工厂中定义一个 Map，它的 key 值与缓存享元对象的 key 是一样的，而 value 就是被引用的次数，这样当外部每次获取该享元的时候，就把对应的引用计数取出来加上 1，然后再记录回去。

2. 实现垃圾回收的基本思路

要实现垃圾回收就比较麻烦点，首先要能确定哪些是垃圾？其次是何时回收？还有由谁来回收？如何回收？解决了这些问题，也就实现了垃圾回收。

（1）为了确定哪些是垃圾，一个简单的方案是这样的，定义一个缓存对象的配置对象，在这个对象中描述了缓存的开始时间和最长不被使用的时间，这个时候判断是否垃圾的计算公式如下：当前的时间 - 缓存的开始时间≥最长不被使用的时间。当然，每次这个对象被使用的时候，就把那个缓存开始的时间更新为使用时的当前时间，也就是说如果一直有人用的话，这个对象是不会被判断为垃圾的。

（2）何时回收的问题，当然是判断出来是垃圾了就可以回收了。

> 提示　**关键是谁来判断垃圾，还有谁来回收垃圾的问题。一个简单的方案是定义一个内部的线程，这个线程在享元工厂被创建的时候就启动运行。由这个线程每隔一定的时间来循环缓存中所有对象的缓存配置，看看是否是垃圾，如果是垃圾，那就可以启动回收了。**

（3）怎么回收呢？这个比较简单，就是直接从缓存的 Map 对象中删除相应的对象，让这些对象没有引用的地方，那么这些对象就可以等着被虚拟机的垃圾回收来回收了。

3. 代码示例

（1）分析了这么多，还是看代码示例会比较清楚，先看缓存配置对象。示例代码如下：

```
/**
 * 描述享元对象缓存的配置对象
 */
public class CacheConfModel{
    /**
     * 缓存开始计时的开始时间
     */
    private long beginTime;
    /**
     * 缓存对象存放的持续时间，其实是最长不被使用的时间
     */
    private double durableTime;
    /**
     * 缓存对象需要被永久存储，也就是不需要从缓存中删除
     */
```

```
    private boolean forever;
    public boolean isForever() {
        return forever;
    }
    public void setForever(boolean forever) {
        this.forever = forever;
    }
    public long getBeginTime() {
        return beginTime;
    }
    public void setBeginTime(long beginTime) {
        this.beginTime = beginTime;
    }
    public double getDurableTime() {
        return durableTime;
    }
    public void setDurableTime(double durableTime) {
        this.durableTime = durableTime;
    }
}
```

（2）对享元对象的管理工作，是由享元工厂来完成的，因此上面的功能，也集中在享元工厂中来实现，在上一个例子的基础之上，来实现这些功能。改进后的享元工厂相对而言稍复杂一点，大致有如下改变。

- 添加一个 Map，来缓存被共享对象的缓存配置的数据。
- 添加一个 Map，来记录缓存对象被引用的次数。
  为了测试方便，定义了一个常量来描述缓存的持续时间。
- 提供获取某个享元被使用的次数的方法。
- 在获取享元的对象中，就要设置相应的引用计数和缓存设置了，示例采用的是内部默认设置一个缓存设置。其实也可以改造一下获取享元的方法，从外部传入缓存设置的数据。
- 提供一个清除缓存的线程，实现判断缓存数据是否已经是垃圾了，如果是，那就把它从缓存中清除掉。

基本上重新实现了享元工厂。示例代码如下：

```
/**
 * 享元工厂，通常实现成为单例
 * 加入实现垃圾回收和引用计数的功能
 */
public class FlyweightFactory {
    private static FlyweightFactory factory =
```

```
                                          new FlyweightFactory();
    private FlyweightFactory(){
        //启动清除缓存值的线程
        Thread t = new ClearCache();
        t.start();
    }
    public static FlyweightFactory getInstance(){
        return factory;
    }

    /**
     * 缓存多个Flyweight对象
     */
    private Map<String,Flyweight> fsMap =
                            new HashMap<String,Flyweight>();
    /**
     * 用来缓存被共享对象的缓存配置，key值和上面Map的一样
     */
    private Map<String,CacheConfModel> cacheConfMap =
                            new HashMap<String,CacheConfModel>();
    /**
     * 用来记录缓存对象被引用的次数，key值和上面Map的一样
     */
    private Map<String,Integer> countMap =
                            new HashMap<String,Integer>();
    /**
     * 默认保存6秒钟，主要为了测试方便，这个时间可以根据应用的要求来设置
     */
    private final long DURABLE_TIME = 6*1000L;

    /**
     * 获取某个享元被使用的次数
     * @param key 享元的key
     * @return 被使用的次数
     */
    public synchronized int getUseTimes(String key){
        Integer count = countMap.get(key);
        if(count==null){
            count = 0;
        }
```

享元工厂实现成单例，在构造方法中启动清除缓存垃圾的线程

```
        return count;
    }
    /**
     * 获取 key 对应的享元对象
     * @param key 获取享元对象的 key
     * @return key 对应的享元对象
     */
    public synchronized Flyweight getFlyweight(String key) {
        Flyweight f = fsMap.get(key);
        if(f==null){
            f = new AuthorizationFlyweight(key);
            fsMap.put(key,f);
            //同时设置引用计数
            countMap.put(key, 1);

            //同时设置缓存配置数据
            CacheConfModel cm = new CacheConfModel();
            cm.setBeginTime(System.currentTimeMillis());
            cm.setForever(false);
            cm.setDurableTime(DURABLE_TIME);

            cacheConfMap.put(key, cm);
        }else{
            //表示还在使用，那么应该重新设置缓存配置
            CacheConfModel cm = cacheConfMap.get(key);
            cm.setBeginTime(System.currentTimeMillis());
            //设置回去
            this.cacheConfMap.put(key, cm);
            //同时计数加 1
            Integer count = countMap.get(key);
            count++;
            countMap.put(key, count);
        }
        return f;
    }
    /**
     * 删除 key 对应的享元对象，连带清除对应的缓存配置和引用次数的记录，不对外
     * @param key 要删除的享元对象的 key
     */
    private synchronized void removeFlyweight(String key){
```

```
        this.fsMap.remove(key);
        this.cacheConfMap.remove(key);
        this.countMap.remove(key);
    }
    /**
     * 维护清除缓存的线程，内部使用
     */
    private  class ClearCache extends Thread{
        public void run(){
            while(true){
                Set<String> tempSet = new HashSet<String>();
                Set<String> set = cacheConfMap.keySet();
                for(String key : set){
                    CacheConfModel ccm = cacheConfMap.get(key);
                    //比较是否需要清除
                    if((System.currentTimeMillis()
                            - ccm.getBeginTime())
                                >= ccm.getDurableTime()){
                        //可以清除，先记录下来
                        tempSet.add(key);
                    }
                }
                //真正清除
                for(String key : tempSet){
                    FlyweightFactory.getInstance()
                                    .removeFlyweight(key);
                }
                System.out.println("now thread="+fsMap.size()
                                +",fsMap=="+fsMap.keySet());
                //休息1秒钟再重新判断
                try {
                    Thread.sleep(1000L);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

> 提示　**getUseTimes、removeFlyweight 和 getFlyweight 这几个方法是加了同步的，原因是在多线程环境下使用它们，容易出现并发错误，比如一个线程在获取享元对象，而另一个线程在删除这个缓存对象。**

（3）要想看出引用计数的效果来，SecurityMgr 需要进行一些修改，至少不要再缓存数据了，需要直接从享元工厂中获取数据，否则就没有办法准确引用计数了。大致改变如下。

- 去掉了放置登录人员对应权限数据的缓存。
- 不需要实现登录功能，在这个示意程序里面，登录方法已经不用实现任何功能，因此直接去掉。
- 原来通过 map 获取值的地方，直接通过 queryByUser 获取就好了。

示例代码如下：

```
public class SecurityMgr {
    private static SecurityMgr securityMgr = new SecurityMgr();
    private SecurityMgr(){
    }
    public static SecurityMgr getInstance(){
        return securityMgr;
    }
    /**
     * 判断某用户对某个安全实体是否拥有某权限
     * @param user 被检测权限的用户
     * @param securityEntity 安全实体
     * @param permit 权限
     * @return true 表示拥有相应权限，false 表示没有相应权限
     */
    public boolean hasPermit(String user,String securityEntity
                                        ,String permit){
        Collection<Flyweight> col = this.queryByUser(user);
        if(col==null || col.size()==0){
            System.out.println(user+"没有登录或是没有被分配任何权限");
            return false;
        }
        for(Flyweight fm : col){
            if(fm.match(securityEntity, permit)){
                return true;
            }
        }
        return false;
```

```
    }
    /**
     * 从数据库中获取某人所拥有的权限
     * @param user 需要获取所拥有的权限的人员
     * @return 某人所拥有的权限
     */
    private Collection<Flyweight> queryByUser(String user){
        Collection<Flyweight> col = new ArrayList<Flyweight>();

        for(String s : TestDB.colDB){
            String ss[] = s.split(",");
            if(ss[0].equals(user)){
                Flyweight fm = null;
                if(ss[3].equals("2")){
                    //表示是组合
                    fm = new UnsharedConcreteFlyweight();
                    //获取需要组合的数据
                    String tempSs[] = TestDB.mapDB.get(ss[1]);
                    for(String tempS : tempSs){
                        Flyweight tempFm = FlyweightFactory
                                .getInstance().getFlyweight(tempS);
                        //把这个对象加入到组合对象中
                        fm.add(tempFm);
                    }
                }else{
                    fm = FlyweightFactory.getInstance()
                            .getFlyweight(ss[1]+","+ss[2]);
                }
                col.add(fm);
            }
        }
        return col;
    }
}
```

（4）还是写个客户端来试试看，上面的享元工厂能否实现对享元对象的管理，尤其是对于垃圾回收和计数方面的功能。对于垃圾回收的功能不需要新添加任何的测试代码，而对于引用计数的功能，需要写代码来调用才能看到效果。示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception{
        SecurityMgr mgr = SecurityMgr.getInstance();
```

```
        boolean f1 = mgr.hasPermit("张三","薪资数据","查看");
        boolean f2 = mgr.hasPermit("李四","薪资数据","查看");
        boolean f3 = mgr.hasPermit("李四","薪资数据","修改");

        for(int i=0;i<3;i++){
            mgr.hasPermit("张三"+i,"薪资数据","查看");
        }

        //特别提醒：这里查看的引用次数，不是指测试使用的次数，指的是
        //SecurityMgr 的 queryByUser 方法通过享元工厂去获取享元对象的次数
        System.out.println("薪资数据,查看 被引用了"+FlyweightFactory
                .getInstance().getUseTimes("薪资数据,查看")+"次");
        System.out.println("薪资数据,修改 被引用了"+FlyweightFactory
                .getInstance().getUseTimes("薪资数据,修改")+"次");
        System.out.println("人员列表,查看 被引用了"+FlyweightFactory
                .getInstance().getUseTimes("人员列表,查看")+"次");
    }
}
```

进行缓存的垃圾回收功能的是个线程在运行，所以你不终止该线程运行，程序会一直运行下去，运行部分结果如下：

```
薪资数据,查看 被引用了 2 次
薪资数据,修改 被引用了 2 次
人员列表,查看 被引用了 6 次
now thread=3,fsMap==[人员列表,查看, 薪资数据,查看, 薪资数据,修改]
now thread=3,fsMap==[人员列表,查看, 薪资数据,查看, 薪资数据,修改]
now thread=3,fsMap==[人员列表,查看, 薪资数据,查看, 薪资数据,修改]
now thread=3,fsMap==[人员列表,查看, 薪资数据,查看, 薪资数据,修改]
now thread=3,fsMap==[人员列表,查看, 薪资数据,查看, 薪资数据,修改]
now thread=3,fsMap==[人员列表,查看, 薪资数据,查看, 薪资数据,修改]

now thread=0,fsMap==[]
now thread=0,fsMap==[]
```

超过 6 秒没有人使用缓存的对象，就进行垃圾回收

解释一下引用次数是怎么计算出来的，目前实现的引用次数，是通过享元工厂获取一次享元对象就计算一次。那么什么时候会通过享元工厂去获取一次享元对象呢？

那就是一个 hasPermit 的请求，在进行权限判断的时候，会查询 TestDB，然后通过享元工厂去获取一次享元对象。因此最后的结果就是看调用一次 hasPermit，这个用户在 TestDB 中对应哪些数据，这些数据就会被调用一次。具体用上面的示例来说就是：

（1）当运行到 Client 下面这句话的时候：

```
boolean f1 = mgr.hasPermit("张三","薪资数据","查看");
```

根据用户名“张三”到 TestDB 中查找，看他具有哪些权限。根据 TestDB，“张三”这个人员只会影响到“人员列表，查看”，因此以“人员列表，查看”为 key 的享元对象被引用一次，当前次数为 1。

（2）Client 继续运行，到下面这句话的时候：

```
boolean f2 = mgr.hasPermit("李四","薪资数据","查看");
```

同理，根据用户名“李四”到 TestDB 中查找，“李四”这个人员会影响到“人员列表，查看”、“薪资数据，查看”和“薪资数据，修改”，因此以这三个描述为 key 的享元对象都被引用一次。此时“人员列表，查看”对应的享元对象当前被引用次数为 2；“薪资数据，查看”对应的享元对象当前被引用次数为 1；“薪资数据，修改”对应的享元对象当前被引用次数为 1。

（3）Client 继续运行，到下面这句话的时候：

```
boolean f3 = mgr.hasPermit("李四","薪资数据","修改");
```

同理，根据用户名“李四”到 TestDB 中查找，然后计数。结果是：以“人员列表，查看”、“薪资数据，查看”和“薪资数据，修改”为 key 的享元对象都再次被引用一次。此时“人员列表，查看”对应的享元对象当前被引用次数为 3；“薪资数据，查看”对应的享元对象当前被引用次数为 2；“薪资数据，修改”对应的享元对象当前被引用次数为 2。

（4）Client 继续运行，执行那个循环，每次运行都只会影响到以“人员列表，查看”为 key 的享元对象的引用计数，每次增加 1 次，因此，循环 3 次后，以“人员列表，查看”为 key 的享元对象被引用的次数为 3 + 3 = 6 次了。

运行客户端测试，体会一下，你还可以在 Client 中加入让线程休息几秒，然后再运行访问权限的数据，这样的话，这些被使用的数据应该会重新计算开始计时的时间，去试试看。当然休息不要超过 6 秒，超过 6 秒就已经清除了。

20.3.4 享元模式的优缺点

享元模式的优点是：减少对象数量，节省内存空间。

可能有的朋友认为共享对象会浪费空间，但是如果这些对象频繁使用，那么其实是节省空间的。因为占用空间的大小等于每个对象实例占用的大小再乘以数量，对于享元对象来讲，基本上就只有一个实例，大大减少了享元对象的数量，并节省不少的内存空间。

节省的空间取决于以下几个因素：因为共享而减少的实例数目、每个实例本身所占用的空间。假如每个对象实例占用 2 个字节，如果不共享数量是 100 个，而共享后就只有一个了，那么节省的空间约等于（100 − 1）×2 字节。

享元模式的缺点是：维护共享对象，需要额外开销。

如同前面演示的享元工厂，在维护共享对象的时候，如果功能复杂，会有很多额外的开销，比如有一个线程来维护垃圾回收。

20.3.5　思考享元模式

1. 享元模式的本质

> 享元模式的本质：分离与共享。

分离的是对象状态中变与不变的部分，共享的是对象中不变的部分。享元模式的关键之处就在于分离变与不变，把不变的部分作为享元对象的内部状态，而变化部分则作为外部状态，由外部来维护，这样享元对象就能够被共享，从而减少对象数量，并节省大量的内存空间。

理解了这个本质后，在使用享元模式的时候，就会考虑，哪些状态需要分离？如何分离？分离后如何处理？哪些需要共享？如何管理共享的对象？外部如何使用共享的享元对象？是否需要不共享的对象？等等。

把这些问题都思考清楚，找到相应的解决方法，那么享元模式也就应用起来了，可能是标准的应用，也可能是变形的应用，但万变不离其宗。

2. 何时选用享元模式

建议在以下情况中选用享元模式。

- 如果一个应用程序使用了大量的细粒度对象，可以使用享元模式来减少对象数量。
- 如果由于使用大量的对象，造成很大的存储开销，可以使用享元模式来减少对象数量，并节约内存。
- 如果对象的大多数状态都可以转变为外部状态，比如通过计算得到，或是从外部传入等，可以使用享元模式来实现内部状态和外部状态的分离。
- 如果不考虑对象的外部状态，可以用相对较少的共享对象取代很多组合对象，可以使用享元模式来共享对象，然后组合对象来使用这些共享对象。

20.3.6　相关模式

- 享元模式与单例模式
  这两个模式可以组合使用。
  通常情况下，享元模式中的享元工厂可以实现成为单例。另外，享元工厂中缓存的享元对象，都是单实例的，可以看成是单例模式的一种变形控制，在享元工厂中来单例享元对象。
- 享元模式与组合模式
  这两个模式可以组合使用。
  在享元模式中，存在不需要共享的享元实现，这些不需要共享的享元通常是对共享的享元对象的组合对象。也就是说，享元模式通常会和组合模式组合使用，来实现更复杂的对象层次结构。
- 享元模式与状态模式

这两个模式可以组合使用。

可以使用享元模式来共享状态模式中的状态对象。通常在状态模式中，会存在数量很大的、细粒度的状态对象，而且它们基本上都是可以重复使用的，都是用来处理某一个固定的状态的，它们需要的数据通常都是由上下文传入，也就是变化部分都分离出去了，所以可以用享元模式来实现这些状态对象。

- 享元模式与策略模式

  这两个模式可以组合使用。

  可以使用享元模式来实现策略模式中的策略对象。和状态模式一样，在策略模式中也存在大量细粒度的策略对象，它们需要的数据同样是从上下文传入的，所以可以使用享元模式来实现这些策略对象

21.1 场景问题

21.1.1 读取配置文件

考虑这样一个实际的应用：维护系统自定义的配置文件。

几乎每个实际的应用系统都有与应用自身相关的配置文件，这个配置文件是由开发人员根据需要自定义的，系统运行时会根据配置的数据进行相应的功能处理。

系统现有的配置数据很简单，主要是 JDBC 所需要的数据，还有默认读取 Spring 的配置文件。目前系统只需要一个 Spring 的配置文件。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <jdbc>
        <driver-class>驱动类名</driver-class>
        <url>连接数据库的URL</url>
        <user>连接数据库的用户名</user>
        <password>连接数据库的密码</password>
    </jdbc>
    <application-xml>缺省读取的Spring配置的文件名称</application-xml>
</root>
```

现在的功能需求是：如何能够灵活地读取配置文件的内容？

21.1.2 不用模式的解决方案

不就是读取配置文件吗？实现很简单，直接读取并解析 xml 就可以了。读取 xml 的应用包很多，这里都不用，直接采用最基础的 Dom 解析就可以了。另外，读取到 xml 中的值后，后续如何处理，这里也不用管，这里只是实现把配置文件读取并解析出来。

按照这个思路，很快就写出了实现的代码。示例代码如下：

```
/**
 * 读取配置文件
 */
public class ReadAppXml {
    /**
     * 读取配置文件内容
     * @param filePathName 配置文件的路径和文件名
     * @throws Exception
     */
    public void read(String filePathName)throws Exception{
        Document doc = null;
        //建立一个解析器工厂
```

```
DocumentBuilderFactory factory =
                    DocumentBuilderFactory.newInstance();
//获得一个DocumentBuilder对象，这个对象代表了具体的Dom解析器
DocumentBuilder builder=factory.newDocumentBuilder();
//得到一个表示xml文档的Document对象
doc=builder.parse(filePathName);
//去掉xml中作为格式化内容的空白而映射在Dom树中的Text Node对象
doc.normalize();

//获取jdbc的配置值
NodeList jdbc = doc.getElementsByTagName("jdbc");
//只有一个jdbc,获取jdbc中的驱动类的名称
NodeList driverClassNode = ((Element)jdbc.item(0))
                  .getElementsByTagName("driver-class");
String driverClass = driverClassNode.item(0)
                  .getFirstChild().getNodeValue();
System.out.println("driverClass=="+driverClass);
//同理获取url、user、password等的值
NodeList urlNode = ((Element)jdbc.item(0))
                  .getElementsByTagName("url");
String url=urlNode.item(0).getFirstChild().getNodeValue();
System.out.println("url=="+url);

NodeList userNode = ((Element)jdbc.item(0))
                  .getElementsByTagName("user");
String user = userNode.item(0).getFirstChild()
                  .getNodeValue();
System.out.println("user=="+user);

NodeList passwordNode = ((Element)jdbc.item(0))
                  .getElementsByTagName("password");
String password = passwordNode.item(0).getFirstChild()
                  .getNodeValue();
System.out.println("password=="+password);

//获取application-xml
NodeList applicationXmlNode =
            doc.getElementsByTagName("application-xml");
String applicationXml = applicationXmlNode.item(0)
                  .getFirstChild().getNodeValue();
```

```
        System.out.println("applicationXml=="+applicationXml);
    }
}
```

21.1.3 有何问题

看了上面的实现，多简单啊，就是最基本的Dom解析嘛，要是采用其他的开源工具包，比如dom4j、jDom之类的来处理，会更简单，这好像不值得一提呀，真的是这样吗？

> 提示 **请思考一个问题：如果配置文件的结构需要变动呢？仔细想想，就会感觉出问题来了。还是先看例子，然后再来总结这个问题。**

随着开发的深入进行，越来越多可配置的数据被抽取出来，需要添加到配置文件中，比如与数据库的连接配置，就加入了是否需要、是否使用DataSource等配置。除了这些还加入了一些其他需要配置的数据，比如，系统管理员、日志记录方式、缓存线程的间隔时长、默认读取哪些Spring配置文件等。示例代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <database-connection>
        <connection-type>连接数据库的类型，1-用Spring集成的方式
            (也就是不用下面两种方式了)，2-DataSource（就是使用JNDI）
            3-使用JDBC自己来连接数据库
        </connection-type>
        <jndi>DataSource的方式用，服务器数据源的JNDI名称</jndi>
        <jdbc>跟上面一样，省略了</jdbc>
    </database-connection>
    <system-operator>系统管理员ID</system-operator>
    <log>
        <operate-type>记录日志的方式，1-数据库，2-文件</operate-type>
        <file-name>记录日志的文件名称</file-name>
    </log>
    <thread-interval>缓存线程的间隔时长</thread-interval>
    <spring-default>
        <application-xmls>
            <application-xml>
                默认读取的Spring配置的文件名称
            </application-xml>
            <application-xml>
                其他需要读取的Spring配置的文件名称
            </application-xml>
        </application-xmls>
```

```
    </spring-default>
</root>
```

有朋友可能会想，改变一下配置文件，值得大惊小怪吗？对于应用系统开发来讲，这不是经常发生的、很普通的一件事情吗？

的确是这样，改变一下配置文件不是件大事情，但是带来的一系列麻烦也不容忽视，比如，修改了配置文件的结构，那么读取配置文件的程序就需要做出相应的变更；用来封装配置文件数据的数据对象也需要相应的修改；外部使用配置文件的地方，获取数据的地方也会相应变动。

当然在这一系列麻烦中，最让人痛苦的莫过于修改读取配置文件的程序了，有时候几乎是重写。比如，在使用 Dom 读取第一个配置文件，读取默认的 Spring 配置文件的值的时候，可能的片段代码示例如下：

```
//获取application-xml
NodeList applicationXmlNode =
               doc.getElementsByTagName("application-xml");
String applicationXml = applicationXmlNode.item(0)
               .getFirstChild().getNodeValue();
System.out.println("applicationXml=="+applicationXml);
```

但是如果配置文件改成第二个，文件的结构发生了改变，需要读取的配置文件变成多个了，读取的程序也发生了改变，而且 application-xml 节点也不是直接从 doc 下获取了。几乎是完全重写了，此时可能的片段代码示例如下：

```
//先要获取spring-default，再获取application-xmls
//然后才能获取application-xml
NodeList springDefaultNode =
               doc.getElementsByTagName("spring-default");
NodeList appXmlsNode = ((Element)springDefaultNode.item(0))
               .getElementsByTagName("application-xmls");
NodeList appXmlNode = ((Element)appXmlsNode.item(0))
               .getElementsByTagName("application-xml");
//循环获取每个application-xml元素的值
for(int i=0;i<appXmlNode.getLength();i++){
    String applicationXml = appXmlNode.item(i)
               .getFirstChild().getNodeValue();
    System.out.println("applicationXml=="+applicationXml);
}
```

仔细对比上面在 xml 变化前后读取值的代码，你会发现，由于 xml 结构的变化，导致读取 xml 文件内容的代码基本上完全重写了。

问题还不仅仅限于读取元素的值，同样体现在读取属性上。可能有些朋友说可以换

不同的xml解析方式来简化，不是还有Sax解析，实在不行换用其他开源的解决方案。

确实通过使用不同的解析xml的方式是会让程序变得简单点，但是每次xml的结构发生变化过后，或多或少都是需要修改程序中解析xml部分的。

有没有办法解决这个问题呢？也就是当xml的结构发生改变后，能够很方便地获取相应元素或者是属性的值，而不用再去修改解析xml的程序。

21.2 解决方案

21.2.1 使用解释器模式来解决问题

用来解决上述问题的一个合理的解决方案，就是使用解释器模式。那么什么是解释器模式呢？

1. 解释器模式的定义

> 给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

这里的文法，简单点说就是我们俗称的“语法规则”。

2. 应用解释器模式来解决问题的思路

要想解决当xml的结构发生改变后，不用修改解析部分的代码，一个自然的思路就是要把解析部分的代码写成公共的，而且还要是通用的，能够满足各种xml取值的需要，比如，获取单个元素的值、获取多个相同名称的元素的值、获取单个元素的属性的值、获取多个相同名称的元素的属性的值，等等。

> **提示** **要写成通用的代码，又有几个问题要解决，如何组织这些通用的代码？如何调用这些通用的代码？以何种方式来告诉这些通用代码，客户端的需要？**

要解决这些问题，其中的一个解决方案就是解释器模式。在描述这个模式的解决思路之前，先解释两个概念，一个是解析器（不是指xml的解析器），另一个是解释器。

- 这里的解析器，指的是把描述客户端调用要求的表达式，经过解析，形成一个抽象语法树的程序，不是指xml的解析器。
- 这里的解释器，指的是解释抽象语法树，并执行每个节点对应的功能的程序。

要解决通用解析xml的问题，第一步：需要先设计一个简单的表达式语言，在客户端调用解析程序的时候，传入用这个表达式语言描述的一个表达式，然后把这个表达式通过解析器的解析，形成一个抽象的语法树。

第二步：解析完成后，自动调用解释器来解释抽象语法树，并执行每个节点所对应的功能，从而完成通用的xml解析。

这样一来，每次当xml结构发生了更改，也就是在客户端调用的时候，传入不同的

表达式即可，整个解析 xml 过程的代码都不需要再修改了。

21.2.2　解释器模式的结构和说明

解释器模式的结构如图 21.1 所示。

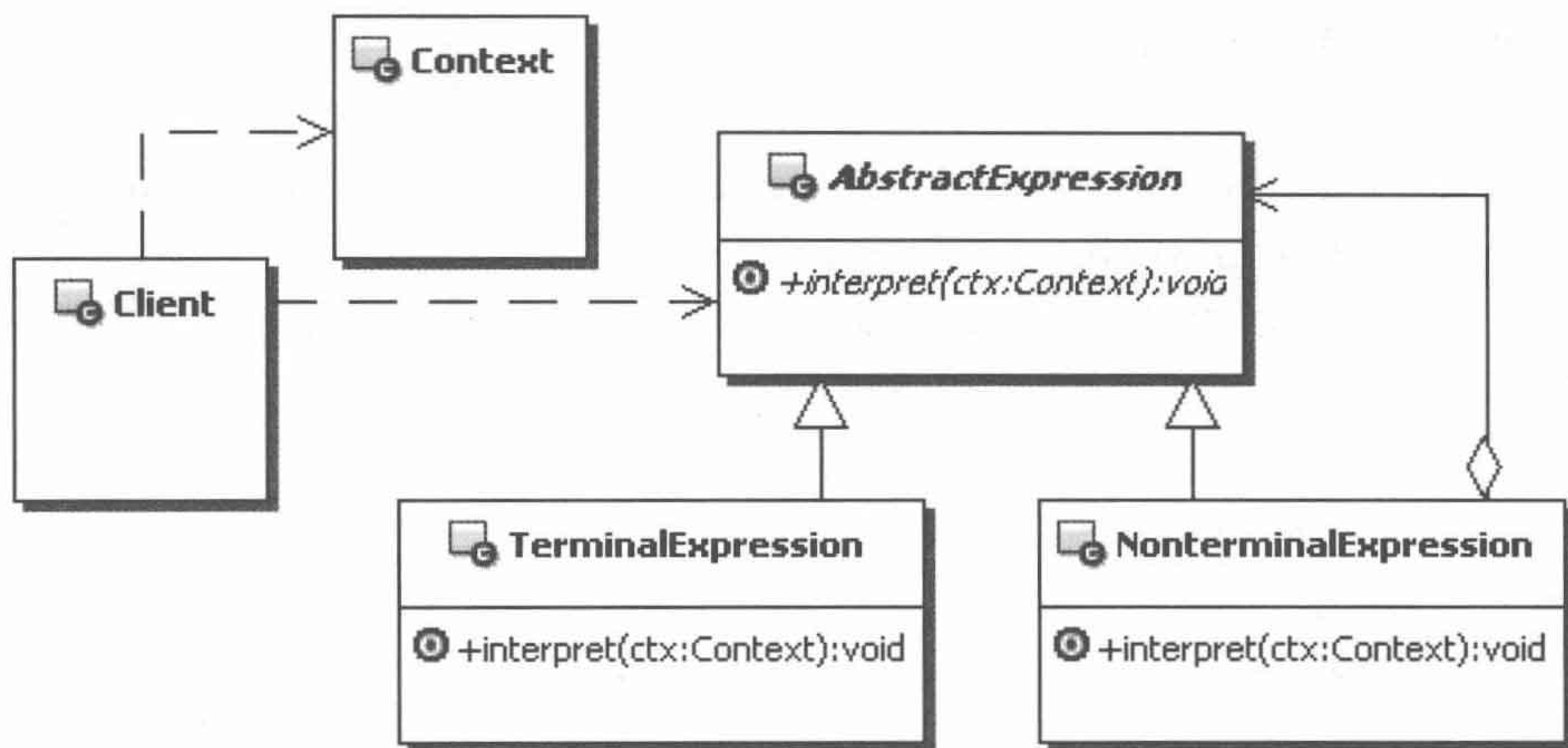


图 21.1　解释器模式结构图

- AbstractExpression：定义解释器的接口，约定解释器的解释操作。
- TerminalExpression：终结符解释器，用来实现语法规则中和终结符相关的操作，不再包含其他的解释器，如果用组合模式来构建抽象语法树的话，就相当于组合模式中的叶子对象，可以有多种终结符解释器。
- NonterminalExpression：非终结符解释器，用来实现语法规则中非终结符相关的操作，通常一个解释器对应一个语法规则，可以包含其他的解释器，如果用组合模式来构建抽象语法树的话，就相当于组合模式中的组合对象。可以有多种非终结符解释器。
- Context：上下文，通常包含各个解释器需要的数据或是公共的功能。
- Client：客户端，指的是使用解释器的客户端，通常在这里将按照语言的语法做的表达式转换成为使用解释器对象描述的抽象语法树，然后调用解释操作。

21.2.3　解释器模式示例代码

（1）先看看抽象表达式的定义。非常简单，定义一个执行解释的方法。示例代码如下：

```
/**
 * 抽象表达式
 */
public abstract class AbstractExpression {
    /**
     * 解释的操作
     * @param ctx 上下文对象
```

```
    */
    public abstract void interpret(Context ctx);
}
```

（2）再来看看终结符表达式的定义。示例代码如下：

```
/**
 * 终结符表达式
 */
public class TerminalExpression extends AbstractExpression{
    public void interpret(Context ctx) {
        //实现与语法规则中的终结符相关联的解释操作
    }
}
```

（3）接下来该看看非终结符表达式的定义了。示例代码如下：

```
/**
 * 非终结符表达式
 */
public class NonterminalExpression extends AbstractExpression{
    public void interpret(Context ctx) {
        //实现与语法规则中的非终结符相关联的解释操作
    }
}
```

（4）上下文的定义。示例代码如下：

```
/**
 * 上下文，包含解释器之外的一些全局信息
 */
public class Context {
}
```

（5）最后来看看客户端的定义。示例代码如下：

```
/**
 * 使用解释器的客户
 */
public class Client {
    //主要按照语法规则对特定的句子构建抽象语法树
    //然后调用解释操作
}
```

看到这里，可能有些朋友会觉得，上面的示例代码里面什么都没有啊。这主要是因为解释器模式是和具体的语法规则联系在一起的，没有相应的语法规则，自然写不出对应的处理代码来。

但是这些示例还是有意义的，可以通过它们看出解释器模式实现的基本架子，只是没有具体的内部处理罢了。

21.2.4　使用解释器模式重写示例

通过上面的讲述可以看出，要使用解释器模式，一个重要的前提就是要定义一套语法规则，也称为文法。不管这套文法的规则是简单还是复杂，必须有这些规则，因为解释器模式就是按照这些规则来进行解析并执行相应的功能的。

1. 为表达式设计简单的文法

为了通用，用 root 表示根元素，a、b、c、d 等来代表元素，一个简单的 xml 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root id="rootId">
    <a>
        <b>
            <c name="testC">12345</c>
            <d id="1">d1</d>
            <d id="2">d2</d>
            <d id="3">d3</d>
            <d id="4">d4</d>
        </b>
    </a>
</root>
```

约定表达式的文法如下。

- 获取单个元素的值：从根元素开始，一直到想要获取值的元素，元素中间用“/”分隔，根元素前不加“/”。比如，表达式“root/a/b/c”就表示获取根元素下、a 元素下、b 元素下的 c 元素的值。
- 获取单个元素的属性的值：要获取值的属性一定是表达式的最后一个元素的属性，在最后一个元素后面添加“.”然后再加上属性的名称。比如，表达式“root/a/b/c.name”就表示获取根元素下、a 元素下、b 元素下、c 元素的 name 属性的值。
- 获取相同元素名称的值，当然是多个，要获取值的元素一定是表达式的最后一个元素，在最后一个元素后面添加“$”。比如，表达式“root/a/b/d$”就表示获取根元素下、a 元素下、b 元素下的多个 d 元素的值的集合。
- 获取相同元素名称的属性的值，当然也是多个：要获取属性值的元素一定是表达式的最后一个元素，在最后一个元素后面添加“$”，然后在后面添加“.”然后再加上属性的名称，在属性名称后面也添加“$”。比如，表达式“root/a/b/d$.id$”就表示获取根元素下、a 元素下、b 元素下的多个 d 元素的 id 属性的值的集合。

2. 示例说明

为了示例的通用性，就使用上面这个简单的 xml 来实现功能，不去使用前面定义的具体的 xml 了，解决的方法是一样的。

另外一个问题，解释器模式主要解决的是“解释抽象语法树，并执行每个节点所对

应的功能”，并不包含如何从一个表达式转换成为抽象的语法树。因此下面的范例就先来实现解释器模式所要求的功能。至于如何从一个表达式转换成为相应的抽象语法树，后面将会给出一个示例。

对于抽象的语法树这个树状结构，很明显可以使用组合模式来构建。解释器模式把需要解释的对象分成了两大类，一类是节点元素，就是可以包含其他元素的组合元素，比如非终结符元素，对应成为组合模式的 Composite；另一类是终结符元素，相当于组合模式的叶子对象。解释整个抽象语法树的过程，也就是执行相应对象的功能的过程。

比如上面的 xml，对应成为抽象语法树，可能的结构如图 21.2 所示：

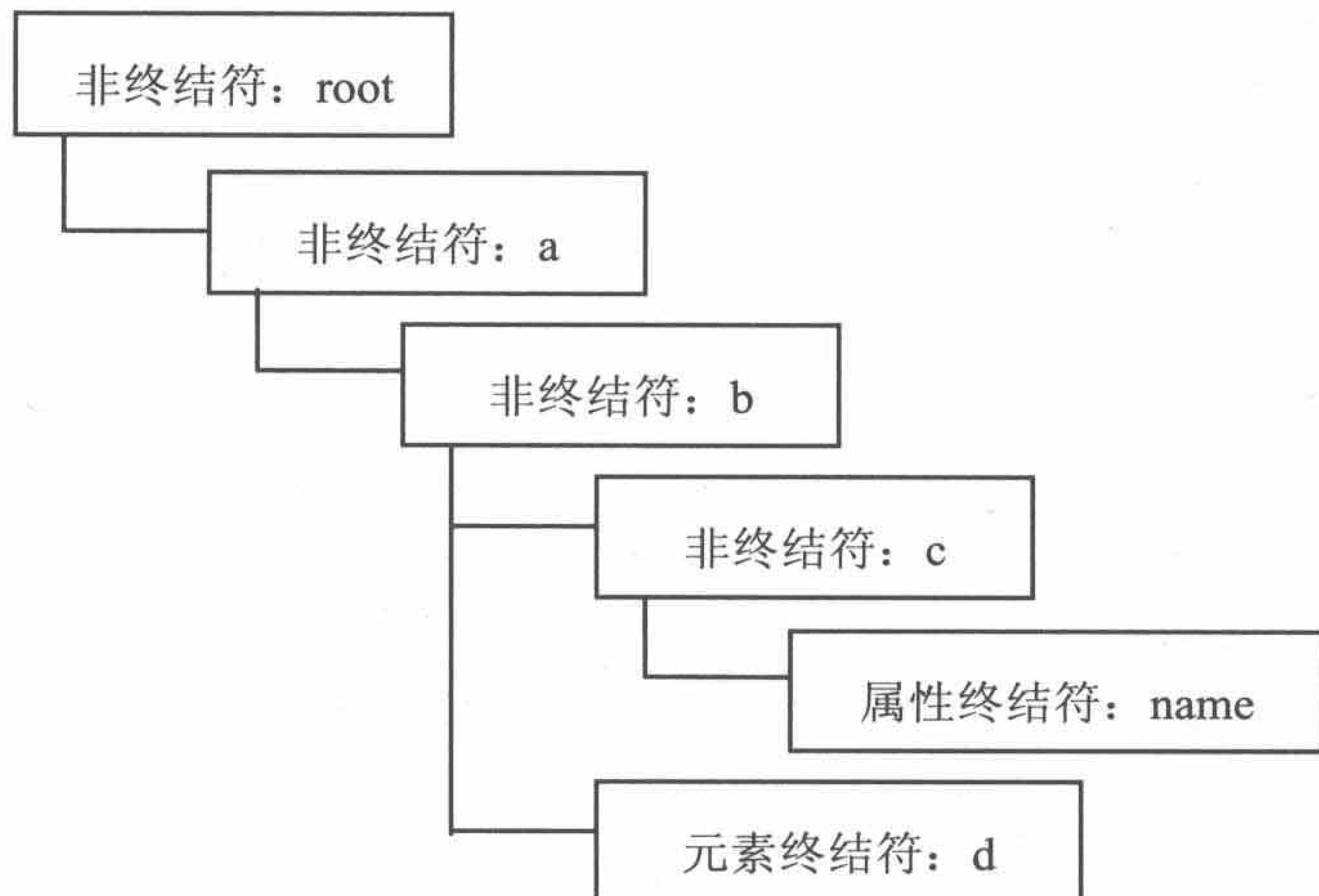


图 21.2　xml 对应的抽象语法树示意图

3. 具体示例

从简单的开始，先来演示获取单个元素的值和单个元素的属性的值。在看具体代码前，先来看看此时系统的整体结构，如图 21.3 所示。

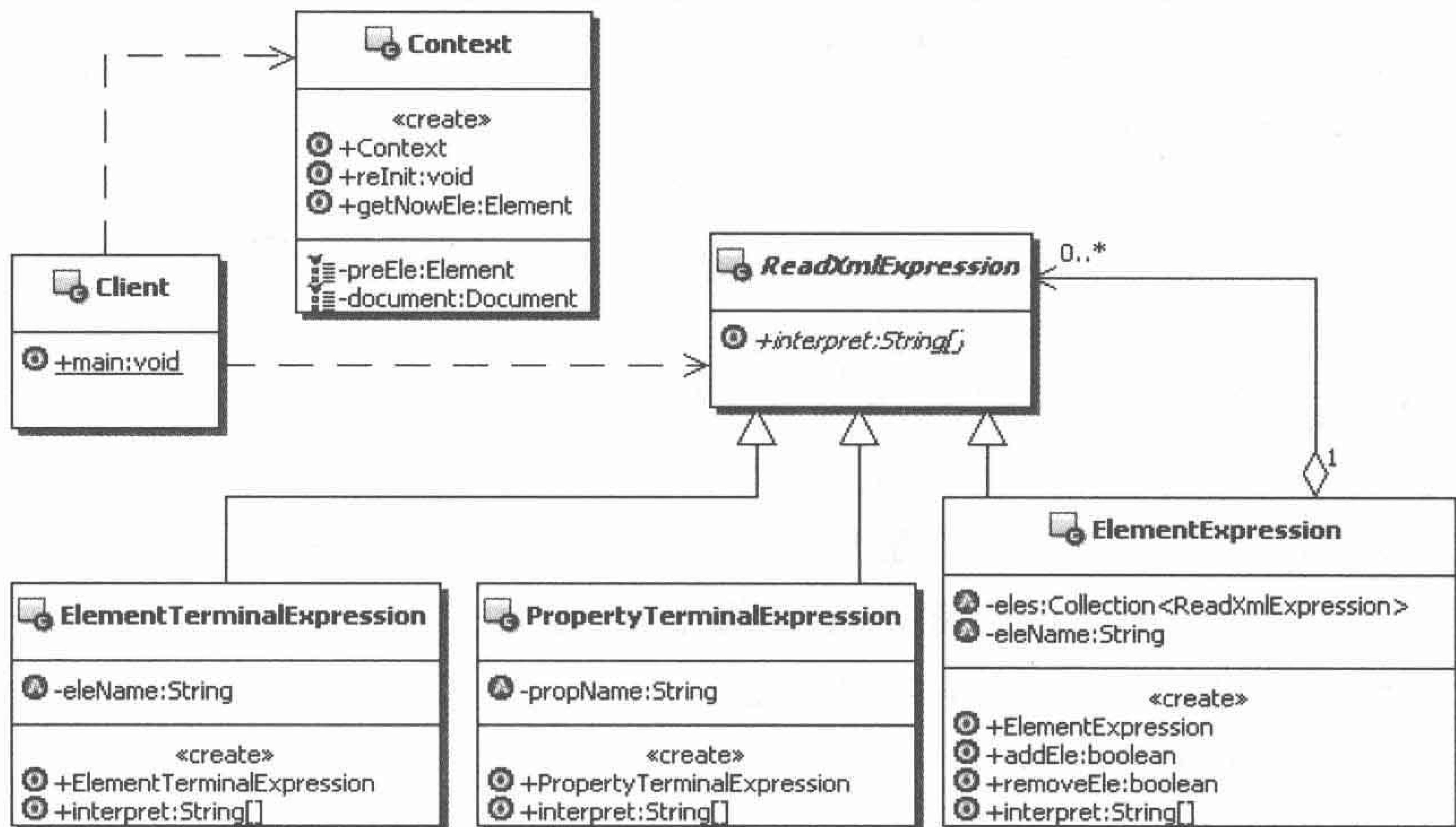


图 21.3　解释器模式示例的结构示意图

（1）定义抽象的解释器。

要实现解释器的功能，首先定义一个抽象的解释器，来约束所有被解释的语法对象，也就是节点元素和终结符元素都要实现的功能。示例代码如下：

```
/**
 * 用于处理自定义 xml 取值表达式的接口
 */
public abstract class ReadXmlExpression {
    /**
     * 解释表达式
     * @param c 上下文
     * @return 解析后的值，为了通用，可能是单个值，也可能是多个值
     *         因此就返回一个数组
     */
    public abstract String[] interpret(Context c);
}
```

（2）定义上下文。

上下文是用来封装解释器需要的一些全局数据，也可以在其中封装一些解释器的公共功能，相当于各个解释器的公共对象。示例代码如下：

```
/**
 *  上下文，用来包含解释器需要的一些全局信息
 */
public class Context {
    /**
     * 上一个被处理的元素
     */
    private Element preEle = null;
    /**
     * Dom 解析 xml 的 Document 对象
     */
    private Document document = null;
    /**
     * 构造方法
     * @param filePathName 需要读取的 xml 的路径和名字
     * @throws Exception
     */
    public Context(String filePathName) throws Exception{
        //通过辅助的 xml 工具类来获取被解析的 xml 对应的 Document 对象
        this.document = XmlUtil.getRoot(filePathName);
    }
/**
     * 重新初始化上下文
     */
    public void reInit(){
```

```
        preEle = null;
    }
    /**
     * 各个 Expression 公共使用的方法
     * 根据父元素和当前元素的名称来获取当前的元素
     * @param pEle 父元素
     * @param eleName 当前元素的名称
     * @return 找到的当前元素
     */
    public Element getNowEle(Element pEle,String eleName){
        NodeList tempNodeList = pEle.getChildNodes();
        for(int i=0;i<tempNodeList.getLength();i++){
            if(tempNodeList.item(i) instanceof Element){
                Element nowEle = (Element)tempNodeList.item(i);
                if(nowEle.getTagName().equals(eleName)){
                    return nowEle;
                }
            }
        }
        return null;
    }

    public Element getPreEle() {
        return preEle;
    }
    public void setPreEle(Element preEle) {
        this.preEle = preEle;
    }

    public Document getDocument() {
        return document;
    }
}
```

在上下文中使用了一个工具对象XmlUtil来获取Document对象，就是Dom解析xml，获取相应的Document对象。示例代码如下：

```
public class XmlUtil {
    public static Document getRoot(String filePathName) throws
Exception{
        Document doc = null;
         //建立一个解析器工厂
```

```
        DocumentBuilderFactory factory =
                        DocumentBuilderFactory.newInstance();
        //获得一个 DocumentBuilder 对象，这个对象代表了具体的 Dom 解析器
        DocumentBuilder builder=factory.newDocumentBuilder();
        //得到一个表示 xml 文档的 Document 对象
        doc=builder.parse(filePathName);
        //去掉xml文档中作为格式化内容的空白而映射在Dom树中的TextNode对象
        doc.normalize();

        return doc;
    }
}
```

（3）定义元素作为非终结符对应的解释器。

接下来该看看如何解释执行中间元素了。首先这个元素相当于组合模式中的 Composite 对象，因此需要对子元素进行维护，另外这个元素的解释处理，只需要把自己找到，作为下一个元素的父元素就可以了。示例代码如下：

```
/**
 * 元素作为非终结符对应的解释器，解释并执行中间元素
 */
public class ElementExpression extends ReadXmlExpression{
    /**
     * 用来记录组合的 ReadXmlExpression 元素
     */
    private Collection<ReadXmlExpression> eles =
                            new ArrayList<ReadXmlExpression>();
    /**
     * 元素的名称
     */
    private String eleName = "";

    public ElementExpression(String eleName){
        this.eleName = eleName;
    }
    public boolean addEle(ReadXmlExpression ele){
        this.eles.add(ele);
        return true;
    }
    public boolean removeEle(ReadXmlExpression ele){
        this.eles.remove(ele);
        return true;
```

对子元素的维护

```
    }
    public String[] interpret(Context c) {
        //先取出上下文中的当前元素作为父级元素
        //查找到当前元素名称所对应的xml元素，并设置回到上下文中
        Element pEle = c.getPreEle();
        if(pEle==null){
            //说明现在获取的是根元素
            c.setPreEle(c.getDocument().getDocumentElement());
        }else{
            //根据父级元素和要查找的元素的名称来获取当前的元素
            Element nowEle = c.getNowEle(pEle, eleName);
            //把当前获取的元素放到上下文中
            c.setPreEle(nowEle);
        }

        //循环调用子元素的interpret方法
        String [] ss = null;
        for(ReadXmlExpression ele : eles){
            ss = ele.interpret(c);
        }
        return ss;
    }
}
```

（4）定义元素作为终结符对应的解释器。

对于单个元素的处理，终结符有两种，一个是元素终结，另一个是属性终结。如果是元素终结，就是要获取元素的值；如果是属性终结，就是要获取属性的值。分别来看看是如何实现的。

先看看元素作为终结的解释器。示例代码如下：

```
/**
 * 元素作为终结符对应的解释器
 */
public class ElementTerminalExpression  extends ReadXmlExpression{
    /**
     * 元素的名字
     */
    private String eleName = "";
    public ElementTerminalExpression(String name){
        this.eleName = name;
    }
```

```
    public String[] interpret(Context c) {
        //先取出上下文中的当前元素作为父级元素
        Element pEle = c.getPreEle();
        //查找到当前元素名称所对应的 xml 元素
        Element ele = null;
        if(pEle==null){
            //说明现在获取的是根元素
            ele = c.getDocument().getDocumentElement();
            c.setPreEle(ele);
        }else{
            //根据父级元素和要查找的元素的名称来获取当前的元素
            ele = c.getNowEle(pEle, eleName);
            //把当前获取的元素放到上下文中
            c.setPreEle(ele);
        }

        //然后需要去获取这个元素的值
        String[] ss = new String[1];
        ss[0] = ele.getFirstChild().getNodeValue();
        return ss;
    }
}
```

（5）定义属性作为终结符对应的解释器。

接下来看看属性终结符的实现，就会比较简单，直接获取最后的元素对象，然后获取相应的属性的值。示例代码如下：

```
/**
 * 属性作为终结符对应的解释器
 */
public class PropertyTerminalExpression extends ReadXmlExpression{
    /**
     * 属性的名字
     */
    private String propName;
    public PropertyTerminalExpression(String propName){
        this.propName = propName;
    }
    public String[] interpret(Context c) {
        //直接获取最后的元素属性的值
        String[] ss = new String[1];
        ss[0] = c.getPreEle().getAttribute(this.propName);
```

```
        return ss;
    }
}
```

（6）使用解释器。

定义好了各个解释器的实现，可以写个客户端来测试一下这些解释器对象的功能了。使用解释器的客户端的工作会比较多，最主要的就是要组装抽象的语法树。

先来看看如何使用解释器获取单个元素的值。示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception {
        //准备上下文
        Context c = new Context("InterpreterTest.xml");

        //想要获取c元素的值，也就是如下表达式的值："root/a/b/c"
        //首先要构建解释器的抽象语法树
        ElementExpression root = new ElementExpression("root");
        ElementExpression aEle = new ElementExpression("a");
        ElementExpression bEle = new ElementExpression("b");
        ElementTerminalExpression cEle =
                            new ElementTerminalExpression("c");
        //组合起来
        root.addEle(aEle);
        aEle.addEle(bEle);
        bEle.addEle(cEle);
        //调用
        String ss[] = root.interpret(c);
        System.out.println("c的值是="+ss[0]);
    }
}
```

把前面定义的 xml 取名叫作“InterpreterTest.xml”，放到当前工程的根下面，运行看看，能正确获取值吗？运行结果如下：

```
c的值是=12345
```

再来测试一下获取单个元素的属性的值。示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception {
        //准备上下文
Context c = new Context("InterpreterTest.xml");

        //想要获取c元素的name属性，也就是如下表达式的值："root/a/b/c.name"
        //这个时候c不是终结了，需要把c修改成ElementExpressioin
        ElementExpression root = new ElementExpression("root");
```

```
        ElementExpression aEle = new ElementExpression("a");
        ElementExpression bEle = new ElementExpression("b");
        ElementExpression cEle = new ElementExpression("c");
        PropertyTerminalExpression prop =
                    new PropertyTerminalExpression("name");
        //组合
        root.addEle(aEle);
        aEle.addEle(bEle);
        bEle.addEle(cEle);
        cEle.addEle(prop);
        //调用
        String ss[] = root.interpret(c);
        System.out.println("c的属性name的值是="+ss[0]);

        //如果要使用同一个上下文，连续进行解析，需要重新初始化上下文对象
        //比如，要连续的重新再获取一次属性name的值，当然你可以重新组合元素，
        //重新解析，只要是在使用同一个上下文，就需要重新初始化上下文对象
        c.reInit();
        String ss2[] = root.interpret(c);
        System.out.println("重新获取c的属性name的值是="+ss2[0]);
    }
}
```

运行结果如下：

```
c的属性name的值是=testC
重新获取c的属性name的值是=testC
```

就像前面讲述的那样，制定一种简单的语言，让客户端用来表达从 xml 中取值的表达式的语言，然后为它们定义一种文法的表示，也就是语法规则，然后用解释器对象来表示那些表达式，接下来通过运行解释器来解释并执行这些功能。

但是从前面的示例中，我们只能看到客户端直接使用解释器对象，来表示客户要从 xml 中取什么值的语法树，而没有看到如何从语言的表达式转换成为这种解释器的表示，这个功能是属于解析器的功能，没有划分在标准的解释器模式中，所以这里就先不演示，在后面将会有示例来介绍解析器。

21.3　模式讲解

21.3.1　认识解释器模式

1. 解释器模式的功能

解释器模式使用解释器对象来表示和处理相应的语法规则，一般一个解释器处理一

条语法规则。理论上来说，只要能用解释器对象把符合语法的表达式表示出来，而且能够构成抽象的语法树，那都可以使用解释器模式来处理。

2. 语法规则和解释器

语法规则和解释器之间是有对应关系的，一般一个解释器处理一条语法规则，但是反过来并不成立，一条语法规则是可以有多种解释和处理的，也就是一条语法规则可以对应多个解释器对象。

3. 上下文的公用性

上下文在解释器模式中起着非常重要的作用。由于上下文会被传递到所有的解释器中，因此可以在上下文中存储和访问解释器的状态，比如，前面的解释器可以存储一些数据在上下文中，后面的解释器就可以获取这些值。

另外还可以通过上下文传递一些在解释器外部，但是解释器需要的数据，也可以是一些全局的、公共的数据。

上下文还有一个功能，就是可以提供所有解释器对象的公共功能，类似于对象组合，而不是使用继承来获取公共功能，在每个解释器对象中都可以调用。

4. 谁来构建抽象语法树

在前面的示例中，大家已经发现，自己在客户端手工构建抽象语法树，是很麻烦的，但是在解释器模式中，并没有涉及这部分功能，只是负责对构建好的抽象语法树进行解释处理。前面的测试简单，所以手工构建抽象语法树也不是特别困难的事，要是复杂了呢？如果还是手工创建，那跟修改解析 xml 的代码也差不了多少。后面会给大家介绍，可以提供解析器来实现把表达式转换成为抽象语法树。

还有一个问题，就是一条语法规则是可以对应多个解释器对象的，也就是说同一个元素，是可以转换成多个解释器对象的，这也就意味着同样一个表达式，是可以构成不同的抽象语法树的，这也造成构建抽象语法树变得很困难，而且工作量非常大。

5. 谁负责解释操作

只要定义好了抽象语法树，肯定是解释器来负责解释执行。虽然有不同的语法规则，但是解释器不负责选择究竟用哪一个解释器对象来解释执行语法规则，选择解释器的功能在构建抽象语法树的时候就完成了。

所以解释器只要忠实地按照抽象语法树解释执行就可以了。

6. 解释器模式的调用顺序示意图

解释器模式的调用顺序如图 21.4 所示。

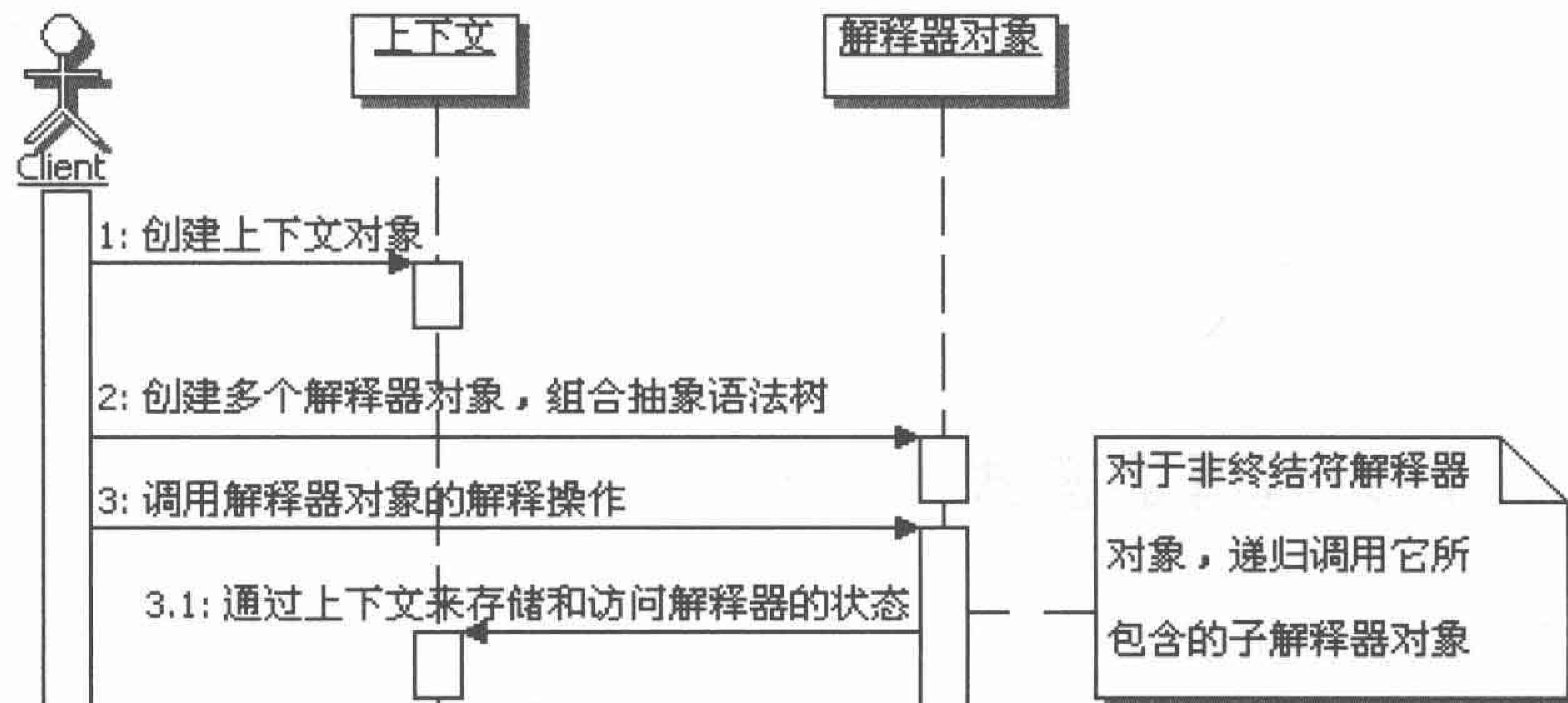


图 21.4　解释器模式的调用顺序示意图

21.3.2　读取多个元素或属性的值

前面介绍了如何获取单个元素的值和单个元素属性的值，下面应该来看看如何获取多个元素的值，还有多个元素中相同名称的属性的值。

获取多个值和前面获取单个值的实现思路大致相同，只是在取值的时候需要循环整个 NodelList，依次取值，而不是只取出第一个来。当然，由于语法发生了变动，所以对应的解释器也需要发生改变。

首先是有了一个表示多个元素作为终结符的语法，比如“root/a/b/d$”中的“d$”；其次有了一个表示多个元素的属性作为终结符的语法，比如“root/a/b/d$.id$”中的“.id$”；最后还有一个表示多个元素，但不是终结符的语法，比如“root/a/b/d$.id$”中的“d$”。

还是看看代码示例吧，会比较清楚。

（1）解释器接口没有变化，原本定义的就是数组，提前做好准备了。

（2）读取 xml 的工具类 XmlUtil 也没有任何变化。

（3）上下文做了一点改变，改变如下。

- 把原来用来记录上一次操作的元素，变成记录上一次操作的多个元素的集合，然后为它提供相应的 getter/setter 方法。
- 原来根据父元素和当前元素的名称获取当前元素的方法，变成了根据父元素和当前元素的名称来获取多个元素。
- 重新初始化上下文的方法里面，初始化的就是记录上一次操作的多个元素的这个集合了。

具体的 Context 类的代码示例如下：

```
/**
 * 上下文，用来包含解释器需要的一些全局信息
 */
public class Context {
    /**
     * Dom 解析 xml 的 Document 对象
     */
    private Document document = null;
    /**
     * 上一次被处理的多个元素
     */
    private List<Element> preEles = new ArrayList<Element>();
    /**
     * 构造方法
     * @param filePathName 需要读取的 xml 的路径和名字
     * @throws Exception
     */
    public Context(String filePathName) throws Exception{
        //通过辅助的 Xml 工具类来获取被解析的 xml 对应的 Document 对象
```

```java
        this.document = XmlUtil.getRoot(filePathName);
    }
    /**
     * 重新初始化上下文
     */
    public void reInit(){
        preEles = new ArrayList<Element>();
    }
    /**
     * 各个 Expression 公共使用的方法
     * 根据父元素和当前元素的名称来获取当前的多个元素的集合
     * @param pEle 父元素
     * @param eleName 当前元素的名称
     * @return 当前的多个元素的集合
     */
    public List<Element> getNowEles(Element pEle,String eleName){
        List<Element> elements = new ArrayList<Element>();
        NodeList tempNodeList = pEle.getChildNodes();
        for(int i=0;i<tempNodeList.getLength();i++){
            if(tempNodeList.item(i) instanceof Element){
                Element nowEle = (Element)tempNodeList.item(i);
                if(nowEle.getTagName().equals(eleName)){
                    elements.add(nowEle);
                }
            }
        }
        return elements;
    }

    public Document getDocument() {
        return document;
    }
    public List<Element> getPreEles() {
        return preEles;
    }
    public void setPreEles(List<Element> nowEles) {
        this.preEles = nowEles;
    }
}
```

（4）处理单个非终结符的对象 ElementExpression，和以前处理单个元素相比，主要

是现在需要面向多个父元素。但是由于是单个非终结符的处理，因此在多个父元素下面去查找符合要求的元素，找到一个就停止。示例代码如下：

```
/**
 * 单个元素作为非终结符的解释器
 */
public class ElementExpression extends ReadXmlExpression{
    /**
     * 用来记录组合的 ReadXmlExpression 元素
     */
    private Collection<ReadXmlExpression> eles =
                              new ArrayList<ReadXmlExpression>();
    /**
     * 元素的名称
     */
    private String eleName = "";
    public ElementExpression(String eleName){
        this.eleName = eleName;
    }
    public boolean addEle(ReadXmlExpression ele){
        this.eles.add(ele);
        return true;
    }
    public boolean removeEle(ReadXmlExpression ele){
        this.eles.remove(ele);
        return true;
    }

    public String[] interpret(Context c) {
        //先取出上下文中的父级元素
        List<Element> pEles = c.getPreEles();
        Element ele = null;
        //把当前获取的元素放到上下文中
        List<Element> nowEles = new ArrayList<Element>();
        if(pEles.size()==0){
            //说明现在获取的是根元素
            ele = c.getDocument().getDocumentElement();
            pEles.add(ele);
            c.setPreEles(pEles);
        }else{
            for(Element tempEle : pEles){
```

```
                nowEles.addAll(c.getNowEles(tempEle, eleName));
                if(nowEles.size()>0){
                    //找到一个就停止
                    break;
                }
            }
            List<Element> tempList = new ArrayList<Element>();
            tempList.add(nowEles.get(0));
            c.setPreEles(tempList);
        }

        //循环调用子元素的 interpret 方法
        String [] ss = null;
        for(ReadXmlExpression tempEle : eles){
            ss = tempEle.interpret(c);
        }
        return ss;
    }
}
```

（5）用来处理单个元素作为终结符的类也发生了一点改变，主要是从多个父元素去获取当前元素，如果当前元素是多个，就取第一个。示例代码如下：

```
/**
 * 元素作为终结符对应的解释器
 */
public class ElementTerminalExpression  extends ReadXmlExpression{
    /**
     * 元素的名字
     */
    private String eleName = "";
    public ElementTerminalExpression(String name){
        this.eleName = name;
    }

    public String[] interpret(Context c) {
        //先取出上下文中的当前元素作为父级元素
        List<Element> pEles = c.getPreEles();
        //查找到当前元素名称所对应的 xml 元素
        Element ele = null;
        if(pEles.size() == 0){
            //说明现在获取的是根元素
```

```
            ele = c.getDocument().getDocumentElement();
        }else{
            //获取当前的元素
            ele = c.getNowEles(pEles.get(0), eleName).get(0);
        }

        //然后来获取这个元素的值
        String[] ss = new String[1];
        ss[0] = ele.getFirstChild().getNodeValue();
        return ss;
    }
}
```

（6）新添加一个解释器，用来解释处理以多个元素的属性作为终结符的情况，它的实现比较简单，只要获取到最后的多个元素对象，然后循环这些元素，一个一个取出相应的属性值就可以了。示例代码如下：

```
/**
 * 以多个元素的属性作为终结符的解释处理对象
 */
public class PropertysTerminalExpression
                                        extends ReadXmlExpression{
    /**
     * 属性名字
     */
    private String propName;
    public PropertysTerminalExpression(String propName){
        this.propName = propName;
    }

    public String[] interpret(Context c) {
        //获取最后的多个元素
        List<Element> eles = c.getPreEles();

        String[] ss = new String[eles.size()];
        //循环多个元素，获取每个元素属性的值
        for(int i=0;i<ss.length;i++){
            ss[i] = eles.get(i).getAttribute(this.propName);
        }
        return ss;
    }
}
```

（7）新添加一个解释器，用来解释处理以多个元素作为终结符的情况。示例代码如下：

```
/**
 * 以多个元素作为终结符的解释处理对象
 */
public class ElementsTerminalExpression  extends ReadXmlExpression{
    /**
     * 元素的名称
     */
    private String eleName = "";
    public ElementsTerminalExpression(String name){
        this.eleName = name;
    }

    public String[] interpret(Context c) {
        //先取出上下文中的父级元素
        List<Element> pEles = c.getPreEles();
        //获取当前的多个元素
        List<Element> nowEles = new ArrayList<Element>();

        for(Element ele : pEles){
            nowEles.addAll(c.getNowEles(ele, eleName));
        }

        //然后来获取这些元素的值
        String[] ss = new String[nowEles.size()];
        for(int i=0;i<ss.length;i++){
            ss[i] = nowEles.get(i).getFirstChild().getNodeValue();
        }
        return ss;
    }
}
```

（8）新添加一个解释器，用来解释处理以多个元素作为非终结符的情况。它的实现类似于以单个元素作为非终结符的情况。只是这次处理的是多个，需要循环处理，同样需要维护子对象。在我们现在设计的语法中，多个元素后面是可以再加子元素的，最起码可以加多个属性的终结符对象。示例代码如下：

```
/**
 * 多个元素作为非终结符的解释处理对象
 */
public class ElementsExpression extends ReadXmlExpression{
```

```
/**
 * 用来记录组合的 ReadXmlExpression 元素
 */
private Collection<ReadXmlExpression> eles =
                        new ArrayList<ReadXmlExpression>();
/**
 * 元素名字
 */
private String eleName = "";
public ElementsExpression(String eleName){
    this.eleName = eleName;
}

public String[] interpret(Context c) {
    //先取出上下文中的父级元素
    List<Element> pEles = c.getPreEles();
    //把当前获取的元素放到上下文中，这次是获取多个元素
    List<Element> nowEles = new ArrayList<Element>();

    for(Element ele : pEles){
        nowEles.addAll(c.getNowEles(ele, eleName));
    }
    c.setPreEles(nowEles);

    //循环调用子元素的 interpret 方法
    String [] ss = null;
    for(ReadXmlExpression ele : eles){
        ss = ele.interpret(c);
    }
    return ss;
}

public boolean addEle(ReadXmlExpression ele){
    this.eles.add(ele);
    return true;
}
public boolean removeEle(ReadXmlExpression ele){
    this.eles.remove(ele);
    return true;
}
```

```
}
```

（9）终于可以写客户端来测试一下了，看看是否能实现期望的功能。先测试获取多个元素值的情况。示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception {
        //准备上下文
        Context c = new Context("InterpreterTest.xml");
        //想要获取多个d元素的值，也就是如下表达式的值："root/a/b/d$"
        //首先要构建解释器的抽象语法树
        ElementExpression root = new ElementExpression("root");
        ElementExpression aEle = new ElementExpression("a");
        ElementExpression bEle = new ElementExpression("b");
        ElementsTerminalExpression dEle =
                                new ElementsTerminalExpression("d");
        //组合起来
        root.addEle(aEle);
        aEle.addEle(bEle);
        bEle.addEle(dEle);
        //调用
        String ss[] = root.interpret(c);
        for(String s : ss){
            System.out.println("d的值是="+s);
        }
    }
}
```

测试结果如下：

```
d的值是=d1
d的值是=d2
d的值是=d3
d的值是=d4
```

接下来测试一下获取多个属性值的情况。示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception {
        //准备上下文
        Context c = new Context("InterpreterTest.xml");

        //想要获取d元素的id属性，也就是如下表达式的值："a/b/d$.id$"
        //首先要构建解释器的抽象语法树
        ElementExpression root = new ElementExpression("root");
        ElementExpression aEle = new ElementExpression("a");
```

```
        ElementExpression bEle = new ElementExpression("b");
        ElementsExpression dEle = new ElementsExpression("d");
        PropertysTerminalExpression prop =
                        new PropertysTerminalExpression("id");
        //组合
        root.addEle(aEle);
        aEle.addEle(bEle);
        bEle.addEle(dEle);
        dEle.addEle(prop);

        //调用
        String ss[] = root.interpret(c);
        for (String s : ss) {
            System.out.println("d的属性id值是=" + s);
        }
    }
}
```

测试结果如下：

```
d 的属性 id 值是=1
d 的属性 id 值是=2
d 的属性 id 值是=3
d 的属性 id 值是=4
```

也很简单吧。只要学会了处理单个的值，处理多个值也就变得容易了，把原来获取单个值的地方改成循环操作即可。

当然，如果要使用同一个上下文进行连续解析，是同样需要重新初始化上下文对象的。

提示	**你还可以尝试一下，如果是想要获取多个元素下的，多个元素的同一个属性的值，能实现吗？自己去测试一下，应该是可以实现的。**

21.3.3　解析器

前面是解释器部分的功能，只要构建好了抽象语法树，解释器就能够正确地解释并执行它，该如何得到这个抽象语法树呢？前面的测试都是人工组合好抽象语法树的，如果实际开发中还这样做，那么上工作量跟修改解析 xml 的代码差不多。

这就需要解析器出场了，这个程序专门负责把按照语法表达的表达式，解析转换成为解释器需要的抽象语法树。当然解析器是和表达式的语法以及解释器对象紧密关联的。

下面来示例一下解析器的实现，把符合前面定义的语法的表达式，转换成为前面实现的解释器的抽象语法树。解析器有很多种实现方式，没有什么定式，只要能完成相应的功能即可，比如表驱动、语法分析生成程序等。这里的示例采用自己来分解表达式以

实现构建抽象语法树的功能，没有使用递归，是采用循环实现的。当然也可以用递归来做。

（1）解析器的实现思路。

要实现解析器也不复杂，大约有以下三个步骤。

① 把客户端传递来的表达式进行分解，分解成为一个一个的元素，并用一个对应的解析模型来封装这个元素的一些信息。

② 根据每个元素的信息，转化成相对应的解析器对象。

③ 按照先后顺序，把这些解析器对象组合起来，就得到抽象语法树了。

可能有朋友会说，为什么不把步骤①和步骤②合并，直接分解出一个元素就转换成相应的解析器对象呢？原因有两个。

- 其一是功能分离，不要让一个方法的功能过于复杂；
- 其二是为了今后的修改和扩展，现在语法简单，所以转换成解析器对象需要考虑的东西少，直接转换也不难，但要是语法复杂了，直接转换就很杂乱了。

事实上，封装解析属性的数据模型充当了步骤①和步骤②操作间的接口，使步骤①和步骤②都变简单了。

（2）下面来看看用来封装每一个解析出来的元素对应的属性对象。示例代码如下：

```
/**
 * 用来封装每一个解析出来的元素对应的属性
 */
public class ParserModel {
    /**
     * 是否单个值
     */
    private boolean singleVlaue;
    /**
     * 是否属性，不是属性就是元素
     */
    private boolean propertyValue;
    /**
     * 是否终结符
     */
    private boolean end;
    public boolean isEnd() {
        return end;
    }
    public void setEnd(boolean end) {
        this.end = end;
    }
    public boolean isSingleVlaue() {
```

```java
        return singleVlaue;
    }
    public void setSingleVlaue(boolean oneVlaue) {
        this.singleVlaue = oneVlaue;
    }
    public boolean isPropertyValue() {
        return propertyValue;
    }
    public void setPropertyValue(boolean propertyValue) {
        this.propertyValue = propertyValue;
    }
}
```

属性相应的 getter/setter

（3）看看解析器的实现，代码稍微复杂点，注释很详尽。为了整体展示解析器，就不去分开每步单讲了。

> **注意** **不过要注意一点，下面这种实现没有考虑并发处理的情况。如果要用在多线程环境下，需要补充相应的处理，特别提示一下。**

示例代码如下：

```java
/**
 * 根据语法来解析表达式，转换成为相应的抽象语法树
 */
public class Parser {
    /**
     * 私有化构造器，避免外部无谓地创建对象实例
     */
    private Parser(){
        //
    }
    //定义几个常量，内部使用
    private final static String BACKLASH = "/";
    private final static String DOT = ".";
    private final static String DOLLAR = "$";
    /**
     * 按照分解的先后记录需要解析的元素的名称
     */
    private static List<String> listEle = null;

    /**
     * 传入一个字符串表达式，通过解析，组合成为一个抽象的语法树
     * @param expr 描述要取值的字符串表达式
```

```
     * @return 对应的抽象语法树
     */
    public static ReadXmlExpression parse(String expr){
        //先初始化记录需解析的元素名称的集合
        listEle = new ArrayList<String>();

        //第一步：分解表达式，得到需要解析的元素名称和该元素对应的解析模型
        Map<String,ParserModel> mapPath = parseMapPath(expr);

        //第二步：根据节点的属性转换成为相应的解释器对象
        List<ReadXmlExpression> list = mapPath2Interpreter(
                                                        mapPath);
        //第三步：组合抽象语法树，一定要按照先后顺序来组合
        //否则对象的包含关系就乱了
        ReadXmlExpression returnRe = buildTree(list);

        return returnRe;
    }
/*----------------------开始实现第一步----------------------*/
    /**
     * 按照从左到右的顺序来分解表达式，得到需要解析的元素名称
     * 还有该元素对应的解析模型
     * @param expr 需要分解的表达式
     * @return 得到需要解析的元素名称，还有该元素对应的解析模型
     */
    private static Map<String,ParserModel> parseMapPath(
                                                String expr){
        //先按照/分割字符串
        StringTokenizer tokenizer = new StringTokenizer(
                                                expr, BACKLASH);
        //初始化一个 Map 用来存放分解出来的值
        Map<String,ParserModel> mapPath =
                                new HashMap<String,ParserModel>();
        while (tokenizer.hasMoreTokens()) {
            String onePath = tokenizer.nextToken();
            if (tokenizer.hasMoreTokens()) {
                //还有下一个值，说明这不是最后一个元素
                //按照现在的语法，属性必然在最后，因此也不是属性
                setParsePath(false,onePath,false,mapPath);
            } else {
```

唯一对外的方法

```
                //说明到最后了
                int dotIndex = onePath.indexOf(DOT);
                if (dotIndex > 0) {
                    //说明是要获取属性的值，那就按照"."来分割
                    //前面的就是元素名字，后面的是属性的名字
                    String eleName = onePath.substring(0, dotIndex);
                    String propName = onePath.substring(dotIndex +
1);
                    //设置属性前面的那个元素，自然不是最后一个，也不是属性
                    setParsePath(false,eleName,false,mapPath);
                    //设置属性，按照现在的语法定义，属性只能是最后一个
                    setParsePath(true,propName,true,mapPath);
                } else {
                    //说明是取元素的值，而且是最后一个元素的值
                    setParsePath(true,onePath,false,mapPath);
                }
                break;
            }
        }
        return mapPath;
    }
    /**
     * 按照分解出来的位置和名称来设置需要解析的元素名称
     * 还有该元素对应的解析模型
     * @param end 是否最后一个
     * @param ele 元素名称
     * @param propertyValue 是否取属性
     * @param mapPath 设置需要解析的元素名称，还有该元素对应的解析模型的 Map
     */
    private static void setParsePath(boolean end,String ele
        ,boolean propertyValue,Map<String,ParserModel> mapPath){
        ParserModel pm = new ParserModel();
        pm.setEnd(end);
        //如果带有$符号就说明不是一个值
        pm.setSingleVlaue(!(ele.indexOf(DOLLAR)>0));
        pm.setPropertyValue(propertyValue);
        //去掉$
        ele = ele.replace(DOLLAR, "");
        mapPath.put(ele,pm);
        listEle.add(ele);
```

```
    }
/*----------------------第一步实现结束----------------------*/

/*----------------------开始实现第二步----------------------*/
    /**
     * 把分解出来的元素名称根据对应的解析模型转换成为相应的解释器对象
     * @param mapPath 分解出来的需解析的元素名称，还有该元素对应的解析模型
     * @return 把每个元素转换成为相应的解释器对象后的集合
     */
    private static List<ReadXmlExpression> mapPath2Interpreter(
                              Map<String,ParserModel> mapPath){
        List<ReadXmlExpression> list =
                              new ArrayList<ReadXmlExpression>();
        //一定要按照分解的先后顺序来转换成解释器对象
        for(String key : listEle){
            ParserModel pm = mapPath.get(key);
            ReadXmlExpression obj = null;
            if(!pm.isEnd()){
                if(pm.isSingleVlaue()){
                    //不是最后一个，是一个值，转化为
                    obj = new ElementExpression(key);
                }else{
                    //不是最后一个，是多个值，转化为
                    obj = new ElementsExpression(key);
                }
            }else{
                if(pm.isPropertyValue()){
                    if(pm.isSingleVlaue()){
                        //是最后一个，是一个值，取属性的值，转化为
                        obj = new PropertyTerminalExpression(key);
                    }else{
                        //是最后一个，是多个值，取属性的值，转化为
                        obj = new PropertysTerminalExpression(key);
                    }
                }else{
                    if(pm.isSingleVlaue()){
                        //是最后一个，是一个值，取元素的值，转化为
                        obj = new ElementTerminalExpression(key);
                    }else{
                        //是最后一个，是多个值，取元素的值，转化为
```

```
                    obj = new ElementsTerminalExpression(key);
                }
            }
        }
        //把转换后的对象添加到集合中
        list.add(obj);
    }
    return list;
}
/*----------------------第二步实现结束----------------------*/

/*----------------------开始实现第三步----------------------*/
private static ReadXmlExpression buildTree(
                              List<ReadXmlExpression> list){
    //第一个对象，也是返回去的对象，就是抽象语法树的根
    ReadXmlExpression returnRe = null;
    //定义上一个对象
    ReadXmlExpression preRe = null;
    for(ReadXmlExpression re : list){
        if(preRe==null){
            //说明是第一个元素
            preRe = re;
            returnRe = re;
        }else{
            //把元素添加到上一个对象下面，同时把本对象设置成为 oldRe
            //作为下一个对象的父节点
            if(preRe instanceof ElementExpression){
                ElementExpression ele =
                              (ElementExpression)preRe;
                ele.addEle(re);
                preRe = re;
            }else if(preRe instanceof ElementsExpression){
                ElementsExpression eles =
                              (ElementsExpression)preRe;
                eles.addEle(re);
                preRe = re;
            }
        }
    }
    return returnRe;
```

```
    }
/*----------------------第三步实现结束-----------------------*/
}
```

（4）看完这个稍长点的解析器程序，该来体会一下，有了它对我们的开发有什么好处？写个客户端来测试看看。现在的客户端就非常简单了，主要有以下三步。

① 首先是设计好想要取值的表达式。

② 然后是通过解析器解析获取抽象语法树。

③ 最后就是请求解释器解释并执行这个抽象语法树，便得到最后的结果了。

客户端测试的示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception {
        //准备上下文
        Context c = new Context("InterpreterTest.xml");
        //通过解析器获取抽象语法树
        ReadXmlExpression re = Parser.parse("root/a/b/d$.id$");
        //请求解析，获取返回值
        String ss[] = re.interpret(c);

        for (String s : ss) {
            System.out.println("d的属性 id值是=" + s);
        }

        //如果要使用同一个上下文，连续进行解析，需要重新初始化上下文对象
        c.reInit();
        ReadXmlExpression re2 = Parser.parse("root/a/b/d$");
        //请求解析，获取返回值
        String ss2[] = re2.interpret(c);
        for (String s : ss2) {
            System.out.println("d的值是=" + s);
        }
    }
}
```

这样就简单多了吧！通过使用解释器模式，自行设计一种简单的语法，就可以用很简单的表达式来获取你想要的 xml 中的值了。有的朋友可能会想到 XPath，没错，本章示例实现的功能就是类似于 XPath 的部分功能。

如果今后 xml 的结构要是发生了变化，或者是想要获取不同的值，基本上就是修改那个表达式而已，你可以试试看，能否完成前面实现过的功能。比如

- 想要获取 c 元素的值，表达式为 “root/a/b/c”；
- 想要获取 c 元素的 name 属性值，表达式为“root/a/b/c.name”；
- 想要获取 d 元素的值，表达式为 “root/a/b/d$”，获取 d 的属性上面已经测试了。

21.3.4　解释器模式的优缺点

解释器模式有以下优点。

- 易于实现语法

  在解释器模式中，一条语法规则用一个解释器对象来解释执行。对于解释器的实现来讲，功能就变得比较简单，只需要考虑这一条语法规则的实现就可以了，其他的都不用管。

- 易于扩展新的语法

  正是由于采用一个解释器对象负责一条语法规则的方式，使得扩展新的语法非常容易。扩展了新的语法，只需要创建相应的解释器对象，在创建抽象语法树的时候使用这个新的解释器对象就可以了。

解释器模式的缺点是不适合复杂的语法。

如果语法特别复杂，构建解释器模式需要的抽象语法树的工作是非常艰巨的，再加上有可能会需要构建多个抽象语法树。所以解释器模式不太适合于复杂的语法，对于复杂的语法，使用语法分析程序或编译器生成器可能会更好一些。

21.3.5　思考解释器模式

1. 解释器模式的本质

解释器模式的本质：**分离实现，解释执行。**

解释器模式通过一个解释器对象处理一个语法规则的方式，把复杂的功能分离开；然后选择需要被执行的功能，并把这些功能组合成为需要被解释执行的抽象语法树；再按照抽象语法树来解释执行，实现相应的功能。

认识这个本质对于识别和变形使用解释器模式是很有作用的。从表面上看，解释器模式关注的是我们平时不太用到的自定义语法的处理；但从实质上看，解释器模式的思路仍然是分离、封装、简化，和很多模式是一样的。

比如，可以使用解释器模式模拟状态模式的功能。如果把解释器模式要处理的语法简化到只有一个状态标记，把解释器看成是对状态的处理对象，对同一个表示状态的语法，可以有很多不同的解释器，也就是有很多不同的处理状态的对象，然后在创建抽象语法树的时候，简化成根据状态的标记来创建相应的解释器，不用再构建树了。看看这样简化下来，是不是可以用解释器模拟出状态模式的功能呢？

同理，解释器模式可以模拟实现策略模式的功能、装饰器模式的功能等，尤其是模拟装饰器模式的功能，构建抽象语法树的过程，自然就对应成为组合装饰器的过程。

2. 何时选用解释器模式

建议在以下情况中选用解释器模式。

当有一个语言需要解释执行，并且可以将该语言中的句子表示为一个抽象语法树的时候，可以考虑使用解释器模式。

在使用解释器模式的时候，还有两个特点需要考虑，一个是语法相对应该比较简单，太复杂的语法不适合使用解释器模式；另一个是效率要求不是很高，对效率要求很高的情况下，不适合使用解释器模式。

21.3.6 相关模式

- 解释器模式和组合模式
  这两个模式可以组合使用。
  通常解释器模式都会使用组合模式来实现，这样能够方便地构建抽象语法树。一般非终结符解释器就相当于组合模式中的组合对象；终结符解释器就相当于叶子对象。
- 解释器模式和迭代器模式
  这两个模式可以组合使用。
  由于解释器模式通常使用组合模式来实现，因此在遍历整个对象结构的时候，自然可以使用迭代器模式。
- 解释器模式和享元模式
  这两个模式可以组合使用。
  在使用解释器模式的时候，可能会造成多个细粒度对象，比如，会有各种各样的终结符解释器，而这些终结符解释器对不同的表达式来说是一样的，是可以共用的，因此可以引入享元模式来共享这些对象。
- 解释器模式和访问者模式
  这两个模式可以组合使用。
  在解释器模式中，语法规则和解释器对象是有对应关系的。语法规则的变动意味着功能的变化，自然会导致使用不同的解释器对象；而且一个语法规则可以被不同的解释器解释执行。
  因此在构建抽象语法树的时候，如果每个节点所对应的解释器对象是固定的，这就意味着该节点对应的功能是固定的，那么就不得不根据需要来构建不同的抽象语法树。
  为了让构建的抽象语法树较为通用，那就要求解释器的功能不要那么固定，要能很方便地改变解释器的功能，这个时候问题就变成了如何能够很方便地更改树形结构中节点对象的功能了，访问者模式可以很好地实现这个功能。

22.1 场 景 问 题

22.1.1 复杂的奖金计算

考虑这样一个实际应用：就是如何实现灵活的奖金计算。

奖金计算是相对复杂的功能，尤其是对于业务部门的奖金计算方式，是非常复杂的，除了业务功能复杂外，还有一个麻烦之处是计算方式还经常需要变动，因为业务部门要通过调整奖金的计算方式来激励士气。

下面从业务上看看现有的奖金计算方式的复杂性。

- 首先是奖金分类，对于个人大致有个人当月业务奖金、个人累计奖金、个人业务增长奖金、及时回款奖金、限时成交加码奖金等；对于业务主管或者是业务经理，除了个人奖金外，还有团队累计奖金、团队业务增长奖金、团队盈利奖金等。
- 其次是计算奖金的金额，又有这么几个基数，销售额、销售毛利、实际回款、业务成本、奖金基数等。
- 另外一个就是计算的公式，针对不同的人、不同的奖金类别、不同的计算奖金的金额，计算的公式是不同的，即使是同一个公式，里面计算的比例参数也有可能是不同的。

22.1.2 简化后的奖金计算体系

看了上面奖金计算的问题，所幸我们只是来学习设计模式，并不是真的要去实现整个奖金计算体系的业务，因此也没有必要把所有的计算业务都罗列在这里。为了后面演示的需要，简化一下。演示用的奖金计算体系如下：

- 每个人当月业务奖金 = 当月销售额×3%
- 每个人累计奖金 = 总的回款额×0.1%
- 团队奖金 = 团队总销售额×1%

22.1.3 不用模式的解决方案

一个人的奖金分成很多个部分。要实现奖金计算，主要就是要按照各个奖金计算的规则，把这个人可以获取的每部分奖金计算出来，然后计算一个总和，这就是这个人可以得到的奖金。

（1）为了演示，先准备点测试数据，在内存中模拟数据库。示例代码如下：

```
/**
 * 在内存中模拟数据库，准备点测试数据，好计算奖金
 */
public class TempDB {
    private TempDB(){
```

```
    }
    /**
     * 记录每个人的月度销售额，只用了人员，月份没有用
     */
    public static Map<String,Double> mapMonthSaleMoney =
                                        new HashMap<String,Double>();
    static{
        //填充测试数据
        mapMonthSaleMoney.put("张三",10000.0);
        mapMonthSaleMoney.put("李四",20000.0);
        mapMonthSaleMoney.put("王五",30000.0);
    }
}
```

（2）按照奖金计算的规则，实现奖金计算。示例代码如下：

```
/**
 * 计算奖金的对象
 */
public class Prize {
    /**
     * 计算某人在某段时间内的奖金，有些参数在演示中并不会使用
     * 但是在实际业务实现上是会用的，为了表示这是个具体的业务方法，
     * 因此这些参数被保留了
     * @param user 被计算奖金的人员
     * @param begin 计算奖金的开始时间
     * @param end 计算奖金的结束时间
     * @return 某人在某段时间内的奖金
     */
    public  double calcPrize(String user,Date begin,Date end){
        double prize = 0.0;
        //计算当月业务奖金，所有人都会计算
        prize = this.monthPrize(user, begin, end);
        //计算累计奖金
        prize += this.sumPrize(user, begin, end);

        //需要判断该人员是普通人员还是业务经理，团队奖金只有业务经理才有
        if(this.isManager(user)){
            prize += this.groupPrize(user, begin, end);
        }
        return prize;
    }
```

```
/**
 * 计算某人的当月业务奖金，参数重复，就不再注释了
 */
private double monthPrize(String user, Date begin, Date end) {
    //计算当月业务奖金，按照人员去获取当月的业务额，然后再乘以 3%
    double prize = TempDB.mapMonthSaleMoney.get(user) * 0.03;
    System.out.println(user+"当月业务奖金"+prize);
    return prize;
}

/**
 * 计算某人的累计奖金，参数重复，就不再注释了
 */
public double sumPrize(String user, Date begin, Date end) {
    //计算累计奖金，其实应该按照人员去获取累计的业务额，然后再乘以 0.1%
    //简单演示一下，假定大家的累计业务额都是 1000000 元
    double prize = 1000000 * 0.001;
    System.out.println(user+"累计奖金"+prize);
    return prize;
}

/**
 * 判断人员是普通人员还是业务经理
 * @param user 被判断的人员
 * @return true 表示是业务经理，false 表示是普通人员
 */
private boolean isManager(String user){
    //应该从数据库中获取人员对应的职务
    //为了演示，简单点判断，只有王五是经理
    if("王五".equals(user)){
        return true;
    }
    return false;
}
/**
 * 计算当月团队业务奖，参数重复，就不再注释了
 */
public double groupPrize(String user, Date begin, Date end) {
    //计算当月团队业务奖金，先计算出团队总的业务额，然后再乘以 1%
```

```
        //假设都是一个团队的
        double group = 0.0;
        for(double d : TempDB.mapMonthSaleMoney.values()){
            group += d;
        }
        double prize = group * 0.01;
        System.out.println(user+"当月团队业务奖金"+prize);
        return prize;
    }
}
```

（3）写个客户端来测试一下，看看是否能正确地计算奖金。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //先创建计算奖金的对象
        Prize p = new Prize();

        //日期对象都没有用上，所以传 null 就可以了
        double zs = p.calcPrize("张三",null,null);
        System.out.println("==========张三应得奖金："+zs);
        double ls = p.calcPrize("李四",null,null);
        System.out.println("==========李四应得奖金："+ls);
        double ww = p.calcPrize("王五",null,null);
        System.out.println("==========王经理应得奖金："+ww);
    }
}
```

测试运行的结果如下：

```
张三当月业务奖金 300.0
张三累计奖金 1000.0
==========张三应得奖金：1300.0
李四当月业务奖金 600.0
李四累计奖金 1000.0
==========李四应得奖金：1600.0
王五当月业务奖金 900.0
王五累计奖金 1000.0
王五当月团队业务奖金 600.0
==========王经理应得奖金：2500.0
```

22.1.4　有何问题

看了上面的实现，觉得挺简单的，就是计算方式麻烦点，每个规则都要实现。真的

很简单吗？仔细想想，有没有什么问题？

对于奖金计算，只是计算方式复杂也就罢了，不过是实现起来会困难点，相对而言还是比较好解决的，不过是用程序把已有的算法表达出来。

最痛苦的是，这些奖金的计算方式经常发生变动，几乎是每个季度都会有小调整，每年都有大调整，这就要求软件的实现要足够灵活，要能够很快进行相应的调整和修改，否则就不能满足实际业务的需要。

举个简单的例子来说，现在根据业务需要，增加一个“环比增长奖金”，就是本月的销售额比上个月有增加，而且要达到一定的比例，当然增长比例越高，奖金比例越大。那么软件就必须要重新实现这个功能，并正确地添加到系统中去。过了两个月，业务奖励的策略发生了变化，不再需要这个奖金了，或者是另外换了一个新的奖金方式了，那么软件就需要把这个功能从软件中去掉，然后再实现新的功能。

那么上面的要求该如何实现呢？

很明显，一种方案是通过继承来扩展功能；另外一种方案就是到计算奖金的对象里面，添加或者删除新的功能，并在计算奖金的时候，调用新的功能或是不调用某些去掉的功能，这种方案会严重违反开—闭原则。

还有一个问题，就是在运行期间，不同人员参与的奖金计算方式也是不同的。举例来说，如果是业务经理，除了参与个人计算部分外，还要参加团队奖金的计算，这就意味着需要在运行期间动态地来组合需要计算的部分，也就是会有一堆的 if-else。

总结一下，奖金计算面临如下问题。

- 计算逻辑复杂。
- 要有足够灵活性，可以方便地增加或者减少功能。
- 要能动态地组合计算方式，不同的人参与的计算不同。

上面描述的奖金计算的问题，绝对没有任何夸大成分，相反已经简化不少了，还有更多麻烦并没有写上来，毕竟我们的重点在设计模式，而不是业务。

把上面的问题抽象一下，设若有一个计算奖金的对象，现在需要能够灵活地给它增加和减少功能，还需要能够动态地组合功能，每个功能就相当于在计算奖金的某个部分。

现在的问题就是，如何才能够透明地给一个对象增加功能，并实现功能的动态组合？

22.2 解决方案

22.2.1 使用装饰模式来解决问题

用来解决上述问题的一个合理的解决方案，就是使用装饰模式。那么什么是装饰模式呢？

1. 装饰模式的定义

> 动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式比生成子类更为灵活。

2. 应用装饰模式来解决问题的思路

虽然经过简化，业务简单了很多，但是需要解决的问题仍不少，还需要解决：透明地给一个对象增加功能，并实现功能的动态组合。

所谓透明地给一个对象增加功能，换句话说就是要给一个对象增加功能，但是不能让这个对象知道，也就是不能去改动这个对象。而实现了给一个对象透明地增加功能，自然就实现了功能的动态组合，比如，原来的对象有 A 功能，现在透明地给它增加了一个 B 功能，是不是就相当于动态组合了 A 和 B 功能呢。

要想实现透明地给一个对象增加功能，也就是要扩展对象的功能，使用继承啊！有人马上提出了一个方案，但很快就被否决了，如果要减少或者修改功能呢？事实上继承是非常不灵活的复用方式。那就用“对象组合”嘛！又有人提出新的方案来了，这个方案得到了大家的赞同。

在装饰模式的实现中，为了能够实现和原来使用被装饰对象的代码无缝结合，是通过定义一个抽象类，让这个类实现与被装饰对象相同的接口，然后在具体实现类中，转调被装饰的对象，在转调的前后添加新的功能，这就实现了给被装饰对象增加功能，这个思路和“对象组合”非常相似。如果对“对象组合”不熟悉，请参见 22.3.1 的第 2 小节。

在转调的时候，如果觉得被装饰对象的功能不再需要了，还可以直接替换掉，也就是不再转调，而是在装饰对象中完成全新的实现。

22.2.2　装饰模式的结构和说明

装饰模式的结构如图 22.1 所示。

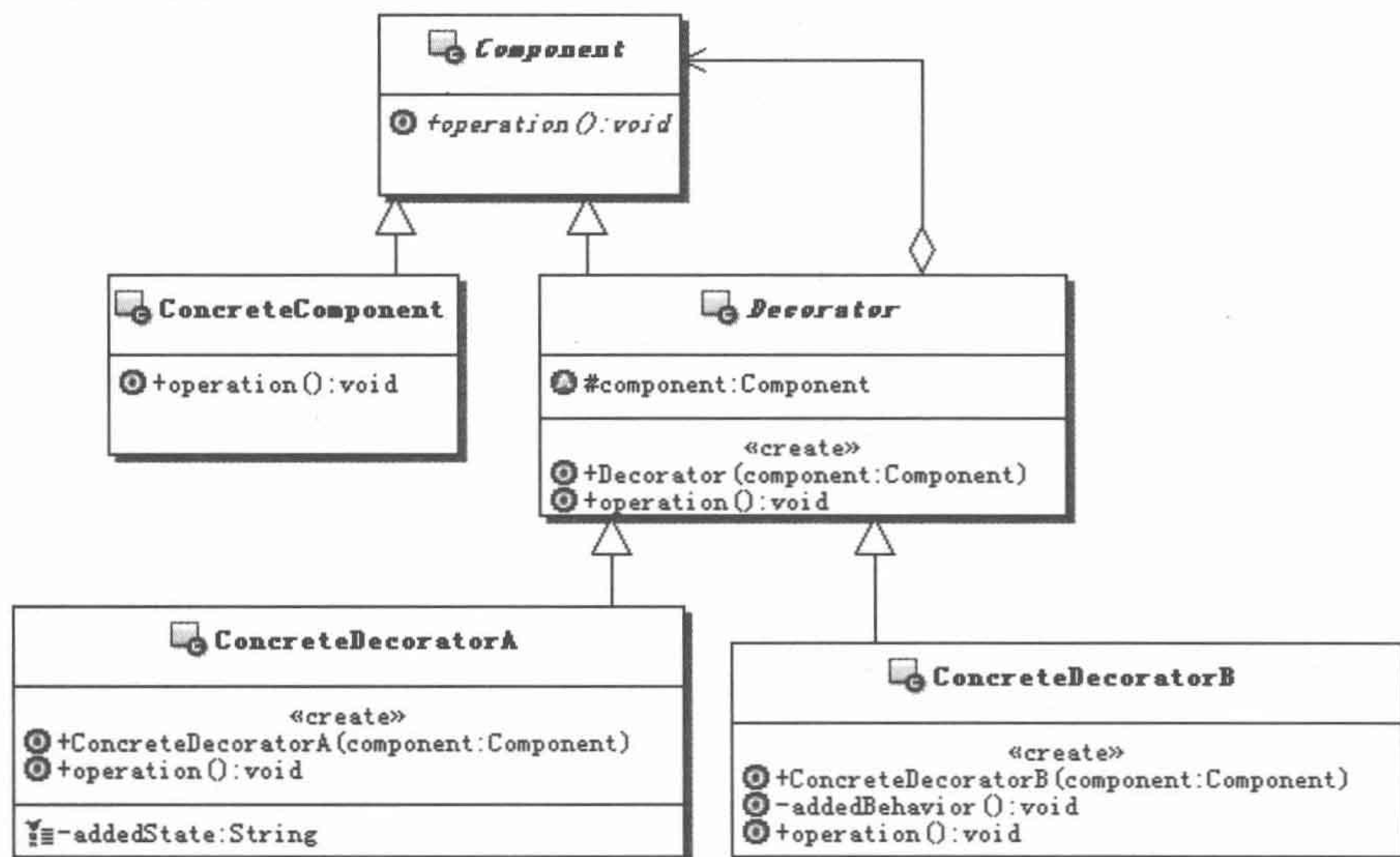


图 22.1　装饰模式的结构图

- Component：组件对象的接口，可以给这些对象动态地添加职责。
- ConcreteComponent：具体的组件对象，实现组件对象接口，通常就是被装饰器装饰的原始对象，也就是可以给这个对象添加职责。

- Decorator：所有装饰器的抽象父类，需要定义一个与组件接口一致的接口，并持有一个 Component 对象，其实就是持有一个被装饰的对象。

> 注意　**注意这个被装饰的对象不一定是最原始的那个对象了，也可能是被其他装饰器装饰过后的对象，反正都是实现的同一个接口，也就是同一类型。**

- ConcreteDecorator：实际的装饰器对象，实现具体要向被装饰对象添加的功能。

22.2.3 装饰模式示例代码

（1）先来看看组件对象的接口定义。示例代码如下：

```
/**
 * 组件对象的接口，可以给这些对象动态地添加职责
 */
public abstract class Component {
    /**
     * 示例方法
     */
    public abstract void operation();
}
```

（2）定义了接口，那就看看具体实现组件对象的示意。示例代码如下：

```
/**
 * 具体实现组件对象接口的对象
 */
public class ConcreteComponent extends Component {
    public void operation() {
        //相应的功能处理
    }
}
```

（3）接下来看看抽象的装饰器对象。示例代码如下：

```
/**
 * 装饰器接口，维持一个指向组件对象的接口对象，并定义一个与组件接口一致的接口
 */
public abstract  class Decorator extends Component {
    /**
     * 持有组件对象
     */
    protected Component component;
    /**
     * 构造方法，传入组件对象
     * @param component 组件对象
```

```
    */
    public Decorator(Component component) {
        this.component = component;
    }
    public void operation() {
        //转发请求给组件对象，可以在转发前后执行一些附加动作
        component.operation();
    }
}
```

（4）该来看看具体的装饰器实现对象了。这里有两个示意对象，一个示意了添加状态，一个示意了添加职责。

先看看添加了状态的示意对象吧。示例代码如下：

```
/**
 * 装饰器的具体实现对象，向组件对象添加职责
 */
public class ConcreteDecoratorA extends Decorator {
    public ConcreteDecoratorA(Component component) {
        super(component);
    }
    /**
     * 添加的状态
     */
    private String addedState;
    public String getAddedState() {
        return addedState;
    }
    public void setAddedState(String addedState) {
        this.addedState = addedState;
    }
    public void operation() {
        //调用父类的方法，可以在调用前后执行一些附加动作
        //在这里进行处理的时候，可以使用添加的状态
        super.operation();
    }
}
```

接下来看看添加职责的示意对象，示例代码如下：

```
/**
 * 装饰器的具体实现对象，向组件对象添加职责
 */
public class ConcreteDecoratorB extends Decorator {
```

```
    public ConcreteDecoratorB(Component component) {
        super(component);
    }
    /**
     * 需要添加的职责
     */
    private void addedBehavior() {
        //需要添加的职责实现
    }
    public void operation() {
        //调用父类的方法，可以在调用前后执行一些附加动作
        super.operation();
        addedBehavior();
    }
}
```

22.2.4 使用装饰模式重写示例

看完了装饰模式的基本知识，接下来考虑如何使用装饰模式重写前面的示例了。要使用装饰模式来重写前面的示例，大致会有以下改变。

- 需要定义一个组件对象的接口，在这个接口中定义计算奖金的业务方法，因为外部就是使用这个接口来操作装饰模式构成的对象结构中的对象。
- 需要添加一个基本的实现组件接口的对象，可以让它返回奖金为 0 就可以了。
- 把各个计算奖金的规则当作装饰器对象，需要为它们定义一个统一的抽象的装饰器对象，方便约束各个具体的装饰器的接口。
- 把各个计算奖金的规则实现成为具体的装饰器对象。

下面看看现在示例的整体结构，以便整体理解和把握示例，如图 22.2 所示。

（1）计算奖金的组件接口和基本的实现对象。

在计算奖金的组件接口中，需要定义原本的业务方法，也就是实现奖金计算的方法。示例代码如下：

```
/**
 * 计算奖金的组件接口
 */
public abstract class Component {
    /**
     * 计算某人在某段时间内的奖金，有些参数在演示中并不会使用
     * 但在实际业务实现上是会用的，为了表示这是个具体的业务方法，
     * 因此这些参数被保留了
     * @param user 被计算奖金的人员
     * @param begin 计算奖金的开始时间
```

```
     * @param end 计算奖金的结束时间
     * @return 某人在某段时间内的奖金
     */
    public abstract double calcPrize(String user
                                    ,Date begin,Date end);
}
```

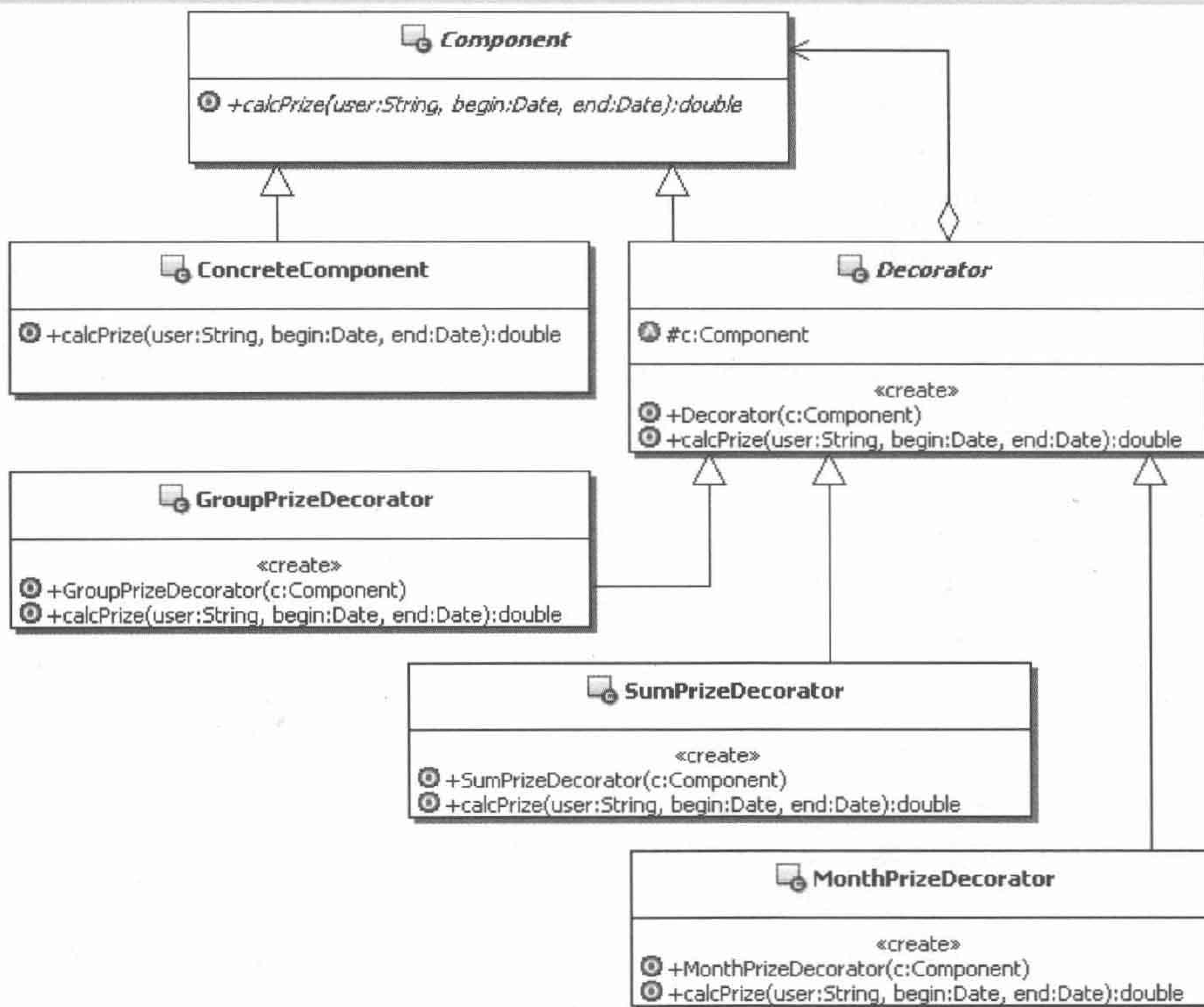


图 22.2　使用装饰模式重写示例的程序结构示意图

为这个业务接口提供一个基本的实现。示例代码如下：

```
/**
 * 基本的实现计算奖金的类，也是被装饰器装饰的对象
 */
public class ConcreteComponent extends Component{
    public double calcPrize(String user, Date begin, Date end) {
        //只是一个默认的实现，默认没有奖金
        return 0;
    }
}
```

（2）定义抽象的装饰器。

在进一步定义装饰器之前，先定义出各个装饰器公共的父类，在这里定义所有装饰器对象需要实现的方法。这个父类应该实现组件的接口，这样才能保证装饰后的对象仍然可以继续被装饰。示例代码如下：

```
/**
 * 装饰器的接口，需要和被装饰的对象实现同样的接口
 */
public abstract class Decorator extends Component{
    /**
     * 持有被装饰的组件对象
     */
    protected Component c;
    /**
     * 通过构造方法传入被装饰的对象
     * @param c 被装饰的对象
     */
    public Decorator(Component c){
        this.c = c;
    }
    public double calcPrize(String user, Date begin, Date end) {
        //转调组件对象的方法
        return c.calcPrize(user, begin, end);
    }
}
```

（3）定义一系列的装饰器对象。

用一个具体的装饰器对象，来实现一条计算奖金的规则。现在有三条计算奖金的规则，那就对应有三个装饰器对象来实现。下面依次来看看它们的实现。

这些装饰器涉及到的 TempDB 和以前一样，这里就不再赘述。

首先来看看实现计算当月业务奖金的装饰器。示例代码如下：

```
/**
 * 装饰器对象，计算当月业务奖金
 */
public class MonthPrizeDecorator extends Decorator{
    public MonthPrizeDecorator(Component c){
        super(c);
    }
    public double calcPrize(String user, Date begin, Date end) {
        //1：先获取前面运算出来的奖金
        double money = super.calcPrize(user, begin, end);
        //2：然后计算当月业务奖金,按人员和时间去获取当月业务额，然后再乘以 3%
        double prize = TempDB.mapMonthSaleMoney.get(user) * 0.03;
        System.out.println(user+"当月业务奖金"+prize);
        return money + prize;
    }
```

```
}
```

接下来看看实现计算累计奖金的装饰器。示例代码如下：

```
/**
 * 装饰器对象，计算累计奖金
 */
public class SumPrizeDecorator extends Decorator{
    public SumPrizeDecorator(Component c){
        super(c);
    }
    public double calcPrize(String user, Date begin, Date end) {
        //1：先获取前面运算出来的奖金
        double money = super.calcPrize(user, begin, end);
        //2：然后计算累计奖金，其实应按人员去获取累计的业务额，然后再乘以 0.1%
        //简单演示一下，假定大家的累计业务额都是 1000000 元
        double prize = 1000000 * 0.001;
        System.out.println(user+"累计奖金"+prize);
        return money + prize;
    }
}
```

接下来看看实现计算当月团队业务奖金的装饰器。示例代码如下：

```
/**
 * 装饰器对象，计算当月团队业务奖金
 */
public class GroupPrizeDecorator extends Decorator{
    public GroupPrizeDecorator(Component c){
        super(c);
    }
    public double calcPrize(String user, Date begin, Date end) {
        //1：先获取前面运算出来的奖金
        double money = super.calcPrize(user, begin, end);
        //2：然后计算当月团队业务奖金，先计算出团队总的业务额，然后再乘以 1%
        //假设都是一个团队的
        double group = 0.0;
        for(double d : TempDB.mapMonthSaleMoney.values()){
            group += d;
        }
        double prize = group * 0.01;
        System.out.println(user+"当月团队业务奖金"+prize);
        return money + prize;
    }
```

```
}
```

（4）使用装饰器的客户端。

使用装饰器的客户端，首先需要创建被装饰的对象，然后创建需要的装饰对象，接下来重要的工作就是组合装饰对象，依次对前面的对象进行装饰。

有很多类似的例子，比如生活中的装修，就拿装饰墙壁来说吧，没有装饰前是原始的砖墙，这就好比是被装饰的对象，首先需要刷腻子，把墙找平，这就好比对原始的砖墙进行了一次装饰，而刷的腻子就好比是一个装饰器对象；好了，装饰一回了，接下来该刷墙面漆了，这又好比装饰了一回，刷的墙面漆就好比是又一个装饰器对象，而且这回被装饰的对象不是原始的砖墙了，而是被腻子装饰器装饰过后的墙面，也就是说后面的装饰器是在前面的装饰器装饰过后的基础之上，继续装饰的，类似于一层一层叠加的功能。

同样的道理，计算奖金也是这样。先创建基本的奖金对象，然后组合需要计算的奖金类型，依次组合计算，最后的结果就是总的奖金。示例代码如下：

```
/**
 * 使用装饰模式的客户端
 */
public class Client {
    public static void main(String[] args) {
        //先创建计算基本奖金的类，这也是被装饰的对象
        Component c1 = new ConcreteComponent();

        //然后对计算的基本奖金进行装饰，这里要组合各个装饰
        //说明，各个装饰者之间最好是不要有先后顺序的限制
        //也就是先装饰谁和后装饰谁都应该是一样的

        //先组合普通业务人员的奖金计算
        Decorator d1 = new MonthPrizeDecorator(c1);
        Decorator d2 = new SumPrizeDecorator(d1);

        //注意：这里只需使用最后组合好的对象调用业务方法即可，会依次调用回去
        //日期对象都没有用上，所以传 null 就可以了
        double zs = d2.calcPrize("张三",null,null);
        System.out.println("==========张三应得奖金："+zs);
        double ls = d2.calcPrize("李四",null,null);
        System.out.println("==========李四应得奖金："+ls);

        //如果是业务经理，还需要一个计算团队的奖金计算
        Decorator d3 = new GroupPrizeDecorator(d2);
        double ww = d3.calcPrize("王五",null,null);
        System.out.println("==========王经理应得奖金："+ww);
```

```
    }
}
```

测试一下，看看结果。示例如下：

```
张三当月业务奖金 300.0
张三累计奖金 1000.0
==========张三应得奖金：1300.0
李四当月业务奖金 600.0
李四累计奖金 1000.0
==========李四应得奖金：1600.0
王五当月业务奖金 900.0
王五累计奖金 1000.0
王五当月团队业务奖金 600.0
==========王经理应得奖金：2500.0
```

当测试运行的时候会按照装饰器的组合顺序，依次调用相应的装饰器来执行业务功能，是一个递归的调用方法，以业务经理“王五”的奖金计算做例子，画个图来说明奖金的计算过程吧，看看是如何调用的，如图 22.3 所示。

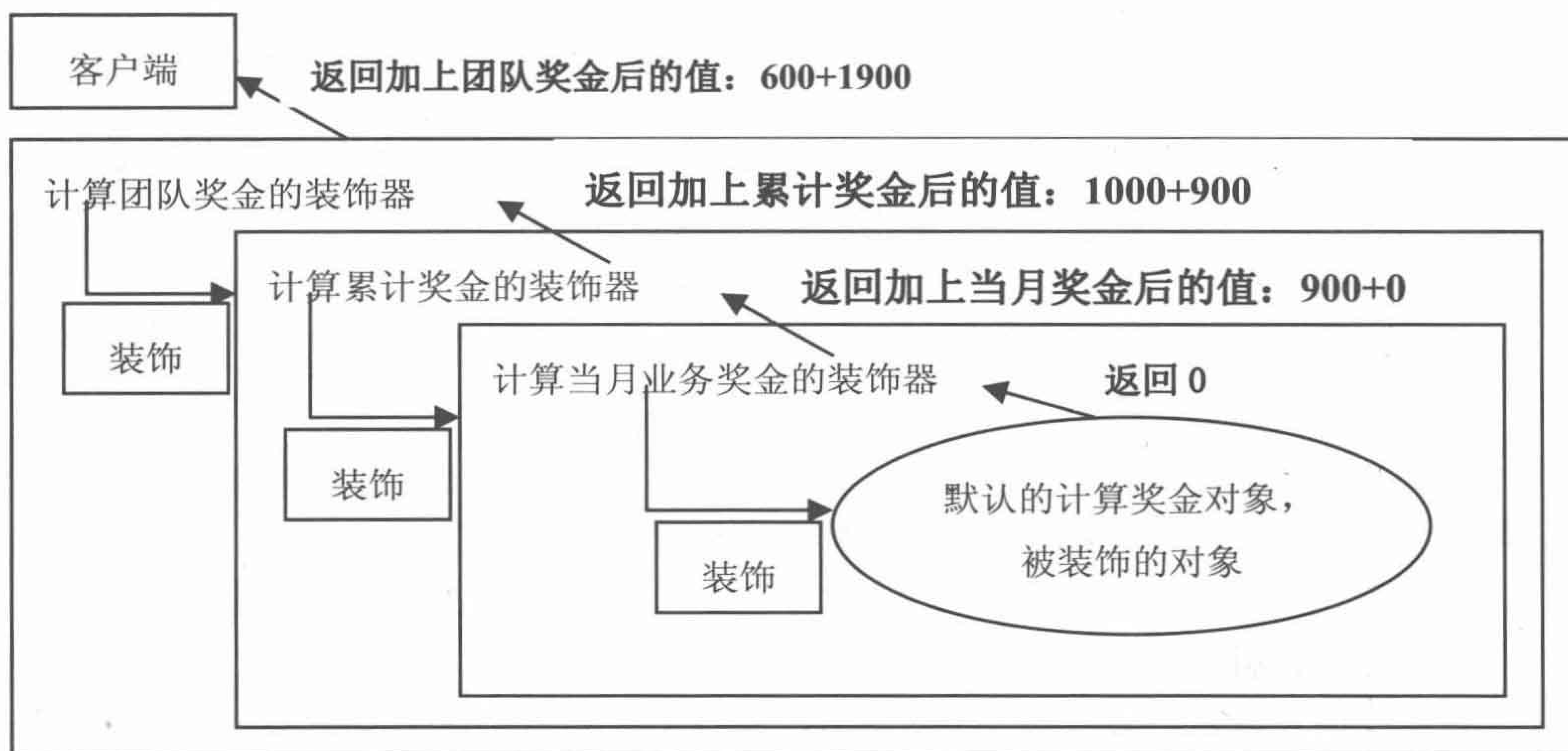


图 22.3　装饰模式示例的组合和调用过程示意图

提示	**这个图很好地揭示了装饰模式的组合和调用过程，请仔细体会一下。**

如同上面的示例，对于基本的计算奖金的对象而言，由于计算奖金的逻辑太过于复杂，而且需要在不同的情况下进行不同的运算，为了灵活性，把多种计算奖金的方式分散到不同的装饰器对象中，采用动态组合的方式，来给基本的计算奖金的对象增添计算奖金的功能，每个装饰器相当于计算奖金的一个部分。

这种方式明显比为基本的计算奖金的对象增加子类更灵活，因为装饰模式的起源点是采用对象组合的方式，然后在组合的时候顺便增加些功能。为了达到一层一层组装的

效果，装饰模式还要求装饰器要实现与被装饰对象相同的业务接口，这样才能以同一种方式依次组合下去。

灵活性还体现在动态上，如果是继承的方式，那么所有的类实例都有这个功能了，而采用装饰模式，可以动态地为某几个对象实例添加功能，而不是对整个类添加功能。比如上面示例中，客户端测试的时候，对张三、李四就只组合了两个功能，对王五就组合了三个功能，但是原始的计算奖金的类都是一样的，只是动态地为它增加的功能不同而已。

22.3 模式讲解

22.3.1 认识装饰模式

1. 装饰模式的功能

装饰模式能够实现动态地为对象添加功能，是从一个对象外部来给对象增加功能，相当于是改变了对象的外观。当装饰过后，从外部使用系统的角度看，就不再是使用原始的那个对象了，而是使用被一系列的装饰器装饰过后的对象。

这样就能够灵活地改变一个对象的功能，只要动态组合的装饰器发生了改变，那么最终所得到的对象的功能也就发生了改变。

变相地还得到了另外一个好处，那就是装饰器功能的复用，可以给一个对象多次增加同一个装饰器，也可以用同一个装饰器装饰不同的对象。

2. 对象组合

前面已经讲到了，一个类的功能的扩展方式，可以是继承，也可以是功能更强大、更灵活的对象组合的方式。

其实，现在**在面向对象的设计中，有一条基本的规则就是“尽量使用对象组合，而不是对象继承”来扩展和复用功能。装饰模式的思考起点就是这个规则。**

可能有些朋友还不太熟悉什么是“对象组合”，下面介绍一下“对象组合”。

什么是对象组合？

直接举例来说吧，假若有一个对象 A，实现了一个 a1 的方法，而 C1 对象想要来扩展 A 的功能，给它增加一个 c11 的方法，那么一个方案是继承，A 对象示例代码如下：

```
public class A {
    public void a1(){
        System.out.println("now in A.a1");
    }
}
```

C1 对象示例代码如下：

```
public class C1 extends A{
    public void c11(){
        System.out.println("now in C1.c11");
    }
```

```
}
```

另外一个方案就是使用对象组合，怎么组合呢？就是在 C1 对象中不再继承 A 对象了，而是去组合使用 A 对象的实例，通过转调 A 对象的功能来实现 A 对象已有的功能。写个新的对象 C2 来示范，示例代码如下：

```
public class C2 {
    /**
     * 创建A对象的实例
     */
    private A a = new A();

    public void a1(){
        //转调A对象的功能
        a.a1();
    }
    public void c11(){
        System.out.println("now in C2.c11");
    }
}
```

大家想想，在转调前后是不是还可以做些功能处理呢？对于 A 对象是不是透明的呢？

对象组合是不是也很简单，而且更灵活了。

- 首先可以有选择地复用功能，不是所有 A 的功能都会被复用，在 C2 中少调用几个 A 定义的功能就可以了；
- 其次在转调前后，可以实现一些功能处理，而且对于 A 对象是透明的，也就是 A 对象并不知道在 a1 方法处理的时候被追加了功能；
- 还有一个额外的好处，就是可以组合拥有多个对象的功能，假如还有一个对象 B，而 C2 也想拥有 B 对象的功能，那很简单，再增加一个方法，然后转调 B 对象就可以了。B 对象示例代码如下：

```
public class B {
    public void b1(){
        System.out.println("now in B.b1");
    }
}
```

同时拥有 A 对象的功能，B 对象的功能，还有自己实现的功能的 C3 对象示例代码如下：

```
public class C3 {
    private A a = new A();
    private B b = new B();

    public void a1(){
        //转调A对象的功能
        a.a1();
```

```
    }
    public void b1(){
        //转调B对象的功能
        b.b1();
    }
    public void c11(){
        System.out.println("now in C3.c11");
    }
}
```


最后再说一点，就是关于对象组合中，**何时创建被组合对象的实例**。

- 一种方案是在属性上直接定义并创建需要组合的对象实例。
- 另外一种方案是在属性上定义一个变量，来表示持有被组合对象的实例，具体实例从外部传入，也可以通过 IoC/DI 容器来注入。

示例代码如下：

```
public class C4 {
    //示例直接在属性上创建需要组合的对象
    private A a = new A();

    //示例通过外部传入需要组合的对象
    private B b = null;
    public void setB(B b){
        this.b = b;
    }
    public void a1(){
        //转调 A 对象的功能
        a.a1();
    }
    public void b1(){
        //转调 B 对象的功能
        b.b1();
    }
    public void c11(){
        System.out.println("now in C4.c11");
    }
}
```

3. 装饰器

装饰器实现了对被装饰对象的某些装饰功能，可以在装饰器中调用被装饰对象的功能，获取相应的值，这其实是一种递归调用。

在装饰器中不仅仅是可以给被装饰对象增加功能，还可以根据需要选择是否调用被装饰对象的功能，如果不调用被装饰对象的功能，那就变成完全重新实现了，相当于动

态修改了被装饰对象的功能。

延伸	**另外一点，各个装饰器之间最好是完全独立的功能，不要有依赖，这样在进行装饰组合的时候，才没有先后顺序的限制，也就是先装饰谁和后装饰谁都应该是一样的，否则会大大降低装饰器组合的灵活性。**

4. 装饰器和组件类的关系

装饰器是用来装饰组件的，装饰器一定要实现和组件类一致的接口，保证它们是同一个类型，并具有同一个外观，这样组合完成的装饰才能够递归调用下去。

组件类是不知道装饰器的存在的，装饰器为组件添加功能是一种透明的包装，组件类毫不知情。需要改变的是外部使用组件类的地方，现在需要使用包装后的类，接口是一样的，但是具体的实现类发生了改变。

5. 退化形式

如果仅仅只是想要添加一个功能，就没有必要再设计装饰器的抽象类了，直接在装饰器中实现跟组件一样的接口，然后实现相应的装饰功能就可以了。但是建议最好还是设计上装饰器的抽象类，这样有利于程序的扩展。

22.3.2　Java 中的装饰模式应用

1. Java 中典型的装饰模式应用——I/O 流

装饰模式在 Java 中最典型的应用，就是 I/O 流，简单回忆一下，如果使用流式操作读取文件内容，会怎样实现呢？简单的代码示例如下：

```
public class IOTest {
    public static void main(String[] args)throws Exception  {
        //流式读取文件
        DataInputStream din = null;
        try{
            din = new DataInputStream(
                new BufferedInputStream(
                        new FileInputStream("IOTest.txt")
                )
            );
            //然后就可以获取文件内容了
            byte bs []= new byte[din.available()];
            din.read(bs);
            String content = new String(bs);
            System.out.println("文件内容===="+content);
        }finally{
            din.close();
        }
    }
```

```
}
```

仔细观察上面的代码，会发现最里层是一个 FileInputStream 对象，然后把它传递给一个 BufferedInputStream 对象，经过 BufferedInputStream 处理，再把处理后的对象传递给了 DataInputStream 对象进行处理，这个过程其实就是装饰器的组装过程，FileInputStream 对象相当于原始的被装饰的对象，而 BufferedInputStream 对象和 DataInputStream 对象则相当于装饰器。

可能有朋友会问，装饰器和具体的组件类是要实现同样的接口的，上面这些类是这样吗？看看 Java 的 I/O 对象层次图吧。由于 Java 的 I/O 对象众多，因此只是画出了 InputStream 的部分，而且由于图的大小关系，也只是表现出了部分的流，具体如图 22.4 所示。

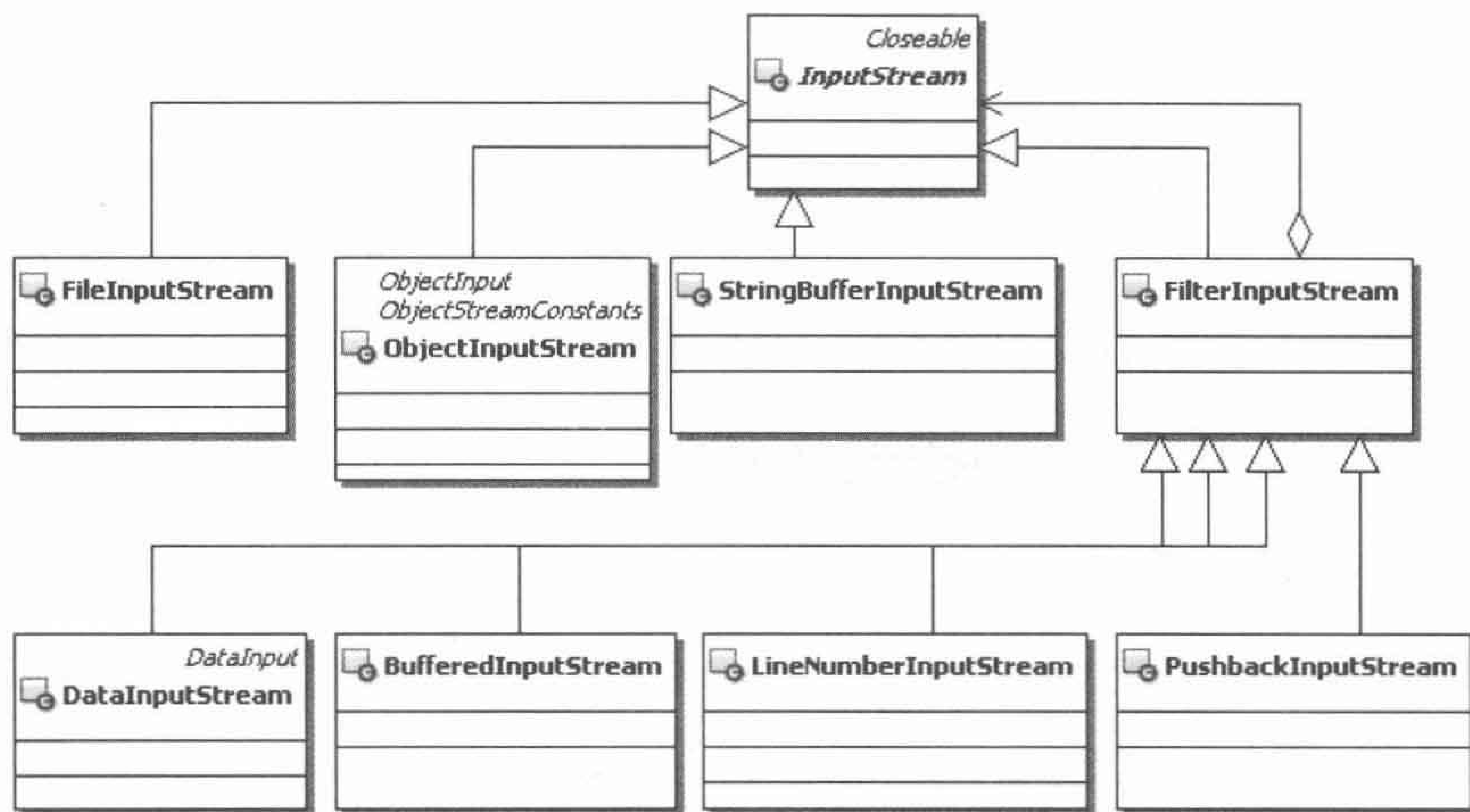


图 22.4　Java 的 I/O 的 InputStream 部分对象层次图

查看图 22.4 会发现，它的结构和装饰模式的结构几乎是一样的。

- InputStream 就相当于装饰模式中的 Component。
- 其实 FileInputStream、ObjectInputStream、StringBufferInputStream 这几个对象是直接继承了 InputSream，还有几个直接继承 InputStream 的对象，比如 ByteArrayInputStream、PipedInputStream 等。这些对象相当于装饰模式中的 ConcreteComponent，是可以被装饰器装饰的对象。
- FilterInputStream 就相当于装饰模式中的 Decorator，而它的子类 DataInputStream、BufferedInputStream、LineNumberInputStream 和 PushbackInputStream 就相当于装饰模式中的 ConcreteDecorator 了。另外 FilterInputStream 和它的子类对象的构造器，都是传入组件 InputStream 类型，这样就完全符合前面讲述的装饰器的结构了。

同样的，输出流部分也类似，就不再赘述。

既然 I/O 流部分是采用装饰模式实现的，如果我们想要添加新的功能，只需要实现新的装饰器，然后在使用的时候，组合进去就可以了。也就是说，我们可以自定义一个装饰器，然后和 JDK 中已有的流的装饰器一起使用。能行吗？试试看吧，前面是按照输入流来讲述的，下面的示例按照输出流来做，顺便体会一下 Java 的输入流和输出流在结构上的相似性。

2. 自己实现的 I/O 流的装饰器——第一版

来个功能简单点的，实现把英文加密存放吧，也谈不上什么加密算法，就是把英文字母向后移动两个位置，比如，a 变成 c，b 变成 d，依次类推，最后的 y 变成 a，z 就变成 b，而且为了简单，只处理小写的，够简单的吧。

好了，还是看看实现简单的加密的代码实现吧。示例代码如下：

```
/**
 * 实现简单的加密
 */
public class EncryptOutputStream  extends OutputStream{
    //持有被装饰的对象
    private OutputStream os = null;
    public EncryptOutputStream(OutputStream os){
        this.os = os;
    }
    public void write(int a) throws IOException {
        //先统一向后移动两位
        a = a+2;
        //97 是小写的 a 的码值
        if(a >= (97+26)){
            //如果大于，表示已经是 y 或者 z 了，减去 26 就回到 a 或者 b 了
            a = a-26;
        }
        this.os.write(a);
    }
}
```

测试一下看看，好用吗？客户端使用代码示例如下：

```
public class Client {
    public static void main(String[] args) throws Exception {
        //流式输出文件
        DataOutputStream dout = new DataOutputStream(
            new BufferedOutputStream(          这是我们加的装饰器
                new EncryptOutputStream(
                    new FileOutputStream("MyEncrypt.txt"))));
        //然后就可以输出内容了
        dout.write("abcdxyz".getBytes());
        dout.close();
    }
}
```

运行一下，打开生成的文件，看看结果。结果示例如下：

```
cdefzab
```

很好，是不是被加密了，虽然是明文的，但已经不是最初存放的内容了，一切显得非常的完美。

再试试看，不是说装饰器可以随意组合吗，换一个组合方式看看，比如把BufferedOutputStream和我们自己的装饰器在组合的时候换个位，示例代码如下：

```
public class Client {
    public static void main(String[] args) throws Exception {
        //流式输出文件
        DataOutputStream dout = new DataOutputStream(
            new EncryptOutputStream (                          换了个位置
                new BufferedOutputStream(
                    new FileOutputStream("MyEncrypt.txt"))));
        dout.write("abcdxyz".getBytes());
        dout.close();
    }
}
```

再次运行，看看结果。坏了，出大问题了，这个时候输出的文件一片空白，什么都没有。这是哪里出了问题呢？

要想把这个问题搞清楚，就需要把上面I/O流的内部运行和基本实现搞明白。分开来看看具体的运行过程吧。

（1）先看看成功输出流中内容的写法的运行过程。

- 当执行到“dout.write("abcdxyz".getBytes());”这句话的时候，会调用DataOutputStream的write方法，把数据输出到BufferedOutputStream中；由于BufferedOutputStream流是一个带缓存的流，它默认缓存8192字节，也就是默认流中的缓存数据到了8192字节，它才会自动输出缓存中的数据；而目前要输出的字节肯定不到8192字节，因此数据就被缓存在BufferedOutputStream流中了，而不会被自动输出。
- 当执行到“dout.close();”这句话的时候，会调用关闭DataOutputStream流，这会转调到传入DataOutputStream中流的close方法，也就是BufferedOutputStream的close方法，而BufferedOutputStream的close方法继承自FilterOutputStream，在FilterOutputStream的close方法实现里面，会先调用输出流的方法flush，然后关闭流。也就是此时BufferedOutputStream流中缓存的数据会被强制输出；BufferedOutputStream流中缓存的数据被强制输出到EncryptOutputStream流，也就是我们自己实现的流，没有缓存，经过处理后继续输出；EncryptOutputStream流会把数据输出到FileOutputStream中，FileOutputStream会直接把数据输出到文件中，因此，这种实现方式会输出文件的内容。

（2）再来看看不能输出流中内容的写法的运行过程。

- 当执行到“dout.write("abcdxyz".getBytes());”这句话的时候，会调用DataOutputStream的write方法，把数据输出到EncryptOutputStream中；EncryptOutputStream流，也就是我们自己实现的流，没有缓存，经过处理后继续输出，把数据输出到BufferedOutputStream中；由于BufferedOutputStream流

是一个带缓存的流，它默认缓存 8192 字节，也就是默认流中的缓存数据到了 8192 字节，它才会自动输出缓存中的数据；而目前要输出的字节肯定不到 8192 字节，因此数据就被缓存在 BufferedOutputStream 流中了，而不会被自动输出。

- 当执行到“dout.close();”这句话的时候：会调用关闭 DataOutputStream 流，这会转调到传入 DataOutputStream 流中的 close 方法，也就是 EncryptOutputStream 的 close 方法，而 EncryptOutputStream 的 close 方法继承自 OutputStream，在 OutputStream 的 close 方法实现中，是个空方法，什么都没有做。因此，这种实现方式没有 flush 流的数据，也就不会输出文件的内容，自然是一片空白了。

3. 自己实现的 I/O 流的装饰器——第二版

要让我们写的装饰器和其他 Java 中的装饰器一样使用，最合理的方案就是：让我们的装饰器继承装饰器的父类，也就是 FilterOutputStream 类，然后使用父类提供的功能来协助完成想要装饰的功能。示例代码如下：

```
public class EncryptOutputStream2  extends FilterOutputStream{
    private OutputStream os = null;
    public EncryptOutputStream2(OutputStream os){
        //调用父类的构造方法
        super(os);
    }
    public void write(int a) throws IOException {
        //先统一向后移动两位
        a = a+2;
        //97 是小写 a 的码值
        if(a >= (97+26)){
            //如果大于，表示已经是 y 或者 z 了，减去 26 就回到 a 或者 b 了
            a = a-26;
        }
        //调用父类的方法
        super.write(a);
    }
}
```

再测试看看，是不是跟其他的装饰器一样，可以随便换位了呢？

22.3.3　装饰模式和 AOP

装饰模式和 AOP 在思想上有共同之处。可能有些朋友还不太了解 AOP，下面先简单介绍一下 AOP 的基础知识。

1. 什么是 AOP——面向方面编程

AOP 是一种编程范式，提供从另一个角度来考虑程序结构以完善面向对象编程（OOP）。

在面向对象开发中，考虑系统的角度通常是纵向的，比如，我们经常画出的如下系

统架构图，默认都是从上到下，上层依赖于下层，如图 22.5 所示。

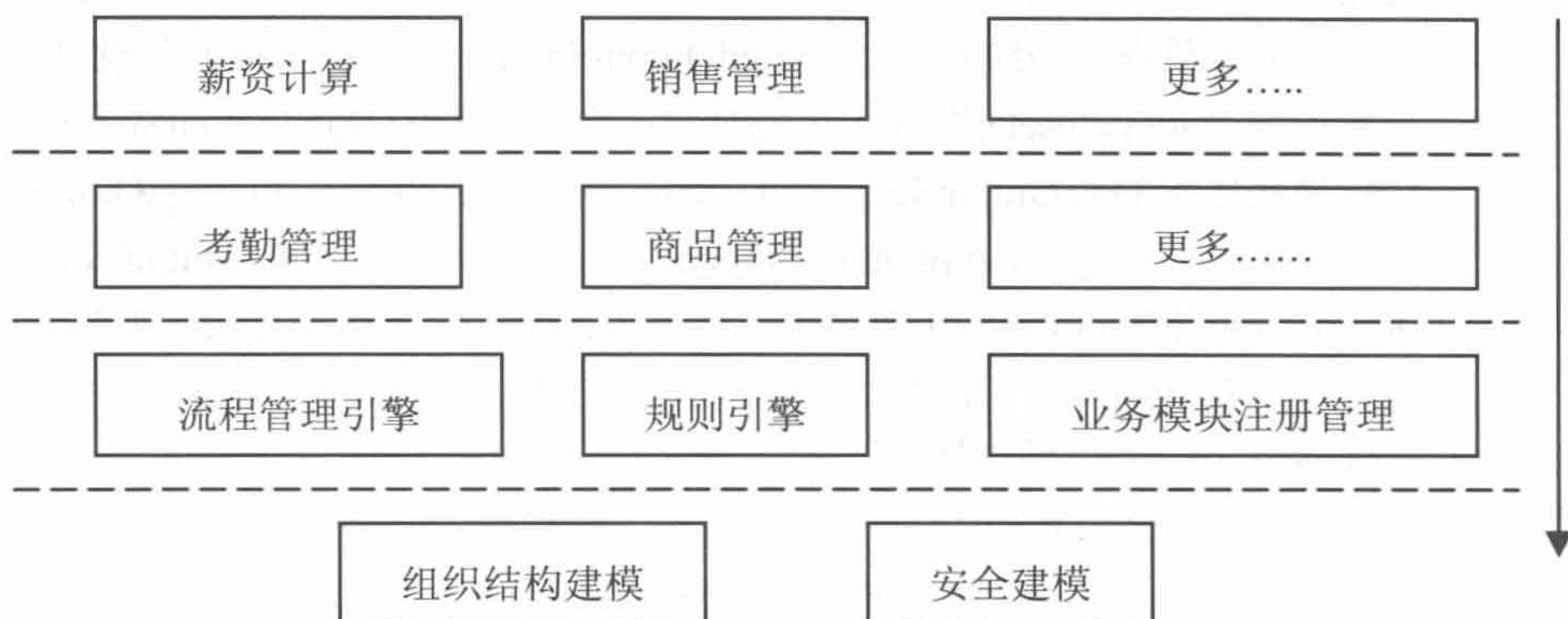


图 22.5　系统架构图示例图

而在每个模块内部呢？就拿大家都熟悉的三层架构来说，也是从上到下来考虑的，通常是表现层调用逻辑层，逻辑层调用数据层，如图 22.6 所示。

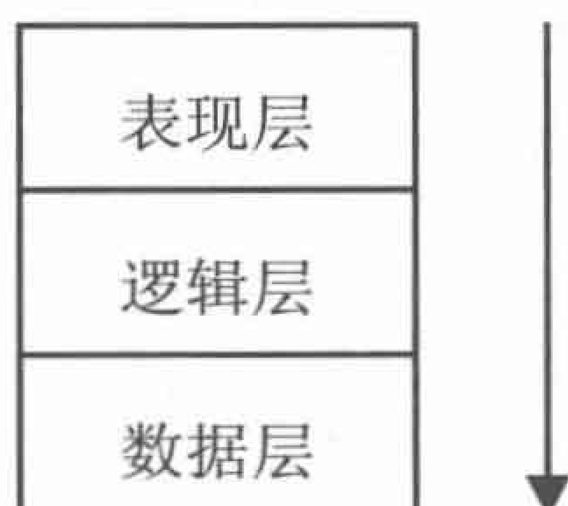


图 22.6　三层架构示意图

慢慢地，越来越多的人发现，在各个模块之中，存在一些共性的功能，比如日志管理、事务管理等，如图 22.7 所示。



图 22.7　共性功能示意图

这个时候，在思考这些共性功能的时候，是从横向来思考问题，与通常面向对象的纵向思考角度不同，很明显，需要有新的解决方案，这个时候 AOP 站出来了。

AOP 为开发者提供了一种描述横切关注点的机制，并能够自动将横切关注点织入到面向对象的软件系统中，从而实现了横切关注点的模块化。

AOP 能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任，比如，事务处理、日志管理、权限控制等，封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

AOP 之所以强大，就是因为它能够自动把横切关注点的功能模块，自动织入回到软件系统中，这是什么意思呢？

先看看没有 AOP，在常规的面向对象系统中，对这种共性的功能如何处理？大都是把这些功能提炼出来，然后在需要用到的地方进行调用如图 22.8 所示，只绘制调用通用日志的公共模块，其他的类似，就不去画了。

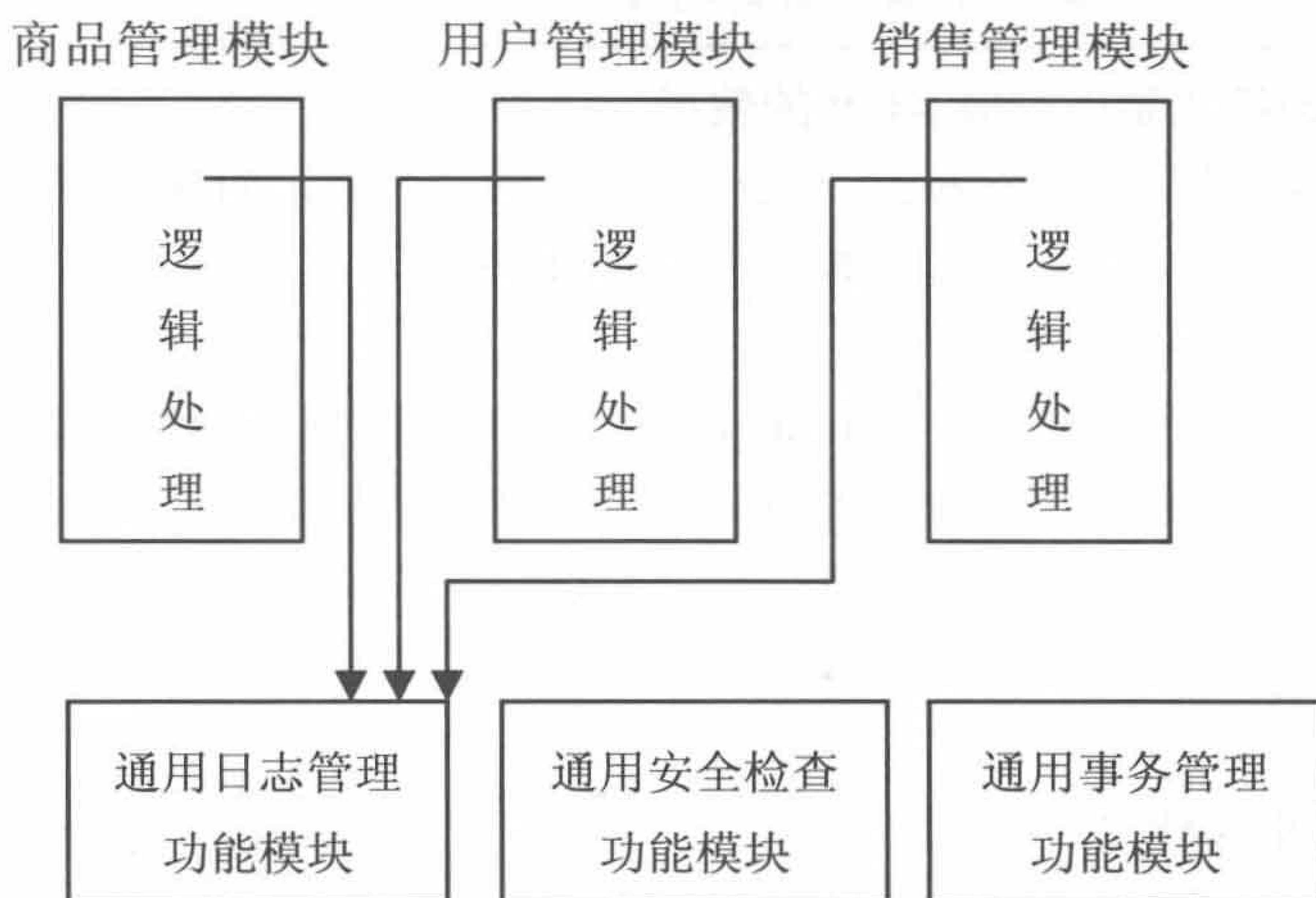


图 22.8 调用公共功能示意图

看清楚，是从应用模块中主动去调用公共模块，也就是应用模块要很清楚公共模块的功能以及具体的调用方法才行，应用模块是依赖于公共模块的，是耦合的，这样一来，要想修改公共模块就会很困难了，牵一而发百。

看看有了 AOP 会怎样？还是画个图来说明，如图 22.9 所示。

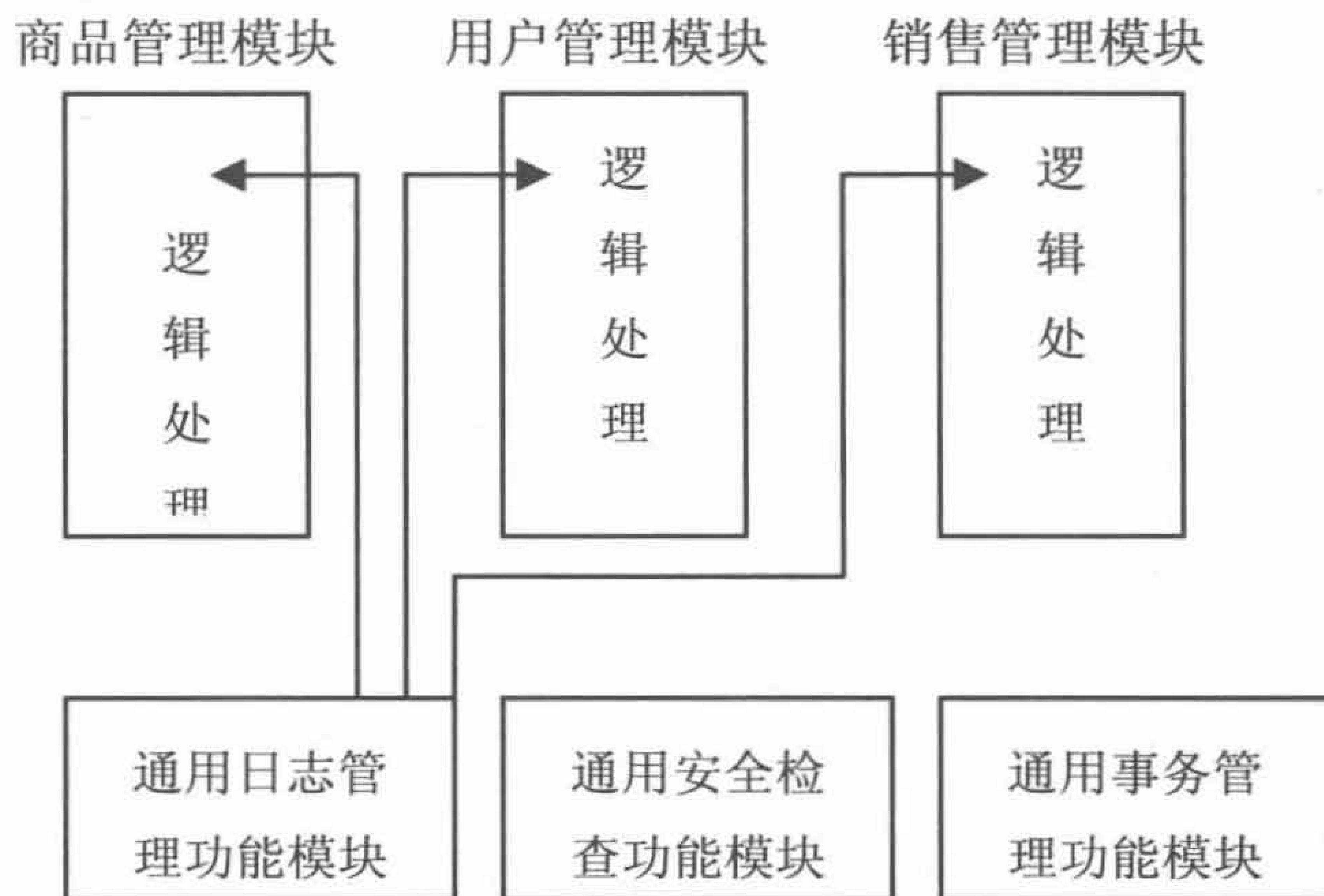


图 22.9 AOP 的调用示意图

乍一看，和上面不用 AOP 没有什么区别嘛，真的吗？看得仔细点，有一个非常非常大的改变，就是**所有的箭头方向反过来了**，原来是应用系统主动去调用各个公共模块的，现在变成了各个公共模块主动织入回到应用系统。

不要小看这一点变化，这样一来应用系统就不需要知道公共功能模块，也就是应用系统和公共模块解耦了。公共功能模块会在合适的时候，由外部织入回到应用系统中，至于谁来实现这样的模块，以及如何实现不在我们的讨论之列，我们更关注其思想。

如果按照装饰模式来对比上述过程，业务功能对象就可以被看做是被装饰的对象，

而各个公共的模块就好比是装饰器，可以透明地来给业务功能对象增加功能。

> 延伸 **所以从某个侧面来说，装饰模式和AOP要实现的功能是类似的，只不过AOP的实现方法不同，会更加灵活，更加可配置，另外AOP一个更重要的变化是思想上的变化——“主从换位”，让原本主动调用的功能模块变成了被动等待，甚至在毫不知情的情况下被织入了很多新的功能。**

2. 使用装饰模式做出类似AOP的效果

下面来演示一下使用装饰模式，把一些公共的功能，比如权限控制、日志记录等透明地添加回到业务功能模块中去，做出类似AOP的效果。

（1）首先定义业务接口。

这个接口相当于装饰模式的Component。注意这里使用的是接口，而不像前面一样使用的是抽象类，虽然使用抽象类的方式来定义组件是装饰模式的标准实现方式，但是如果不需要为子类提供公共功能的话，也是可以实现成接口的，这点要先说明一下，免得有些朋友会认为这就不是装饰模式了。示例代码如下：

```
/**
 * 商品销售管理的业务接口
 */
public interface GoodsSaleEbi {
    /**
     * 保存销售信息，本来销售数据应该是多条，太麻烦了，为了演示，简单点
     * @param user 操作人员
     * @param customer 客户
     * @param saleModel 销售数据
     * @return 是否保存成功
     */
    public boolean sale(String user,String customer,
                                        SaleModel saleModel);
}
```

顺便把封装业务数据的对象也定义出来。很简单，示例代码如下：

```
/**
 * 封装销售单的数据，简单地示意一些
 */
public class SaleModel {
    /**
     * 销售的商品
     */
    private String goods;
    /**
     * 销售的数量
     */
```

```
    private int saleNum;
    public String getGoods() {
       return goods;
    }
    public void setGoods(String goods) {
       this.goods = goods;
    }
    public int getSaleNum() {
        return saleNum;
    }
    public void setSaleNum(int saleNum) {
        this.saleNum = saleNum;
    }
    public String toString(){
        return "商品名称="+goods+",购买数量="+saleNum;
    }
}
```

就是 getter/setter 方法

（2）定义基本的业务实现对象。示例代码如下：

```
public class GoodsSaleEbo implements GoodsSaleEbi{
    public boolean sale(String user,String customer,
                                  SaleModel saleModel) {
        System.out.println(user+"保存了"
                    +customer+"购买 "+saleModel+" 的销售数据");
        return true;
    }
}
```

（3）接下来该来实现公共功能了。把这些公共功能实现成为装饰器，则需要给它们定义一个抽象的父类。示例代码如下：

```
/**
 * 装饰器的接口，需要和被装饰的对象实现同样的接口
 */
public abstract class Decorator implements GoodsSaleEbi{
    /**
     * 持有被装饰的组件对象
     */
    protected GoodsSaleEbi ebi;
    /**
     * 通过构造方法传入被装饰的对象
     * @param ebi 被装饰的对象
     */
```

```
    public Decorator(GoodsSaleEbi ebi){
        this.ebi = ebi;
    }
}
```

（4）实现权限控制的装饰器。

先检查是否有运行的权限，如果有就继续调用，如果没有，就不递归调用了，而是输出没有权限的提示。示例代码如下：

```
/**
 * 实现权限控制
 */
public class CheckDecorator extends Decorator{
    public CheckDecorator(GoodsSaleEbi ebi){
        super(ebi);
    }
    public boolean sale(String user,String customer
                                  , SaleModel saleModel) {
        //简单点，只让张三执行这个功能
        if(!"张三".equals(user)){
            System.out.println("对不起"+user
                                  +"，你没有保存销售单的权限");
            //就不再调用被装饰对象的功能了
            return false;
        }else{
            return this.ebi.sale(user, customer, saleModel);
        }
    }
}
```

（5）实现日志记录的装饰器，就是在功能执行完成后记录日志即可。示例代码如下：

```
/**
 * 实现日志记录
 */
public class LogDecorator extends Decorator{
    public LogDecorator(GoodsSaleEbi ebi){
        super(ebi);
    }
    public boolean sale(String user,String customer,
                                 SaleModel saleModel) {
        //执行业务功能
        boolean f = this.ebi.sale(user, customer, saleModel);

        //在执行业务功能后记录日志
```

```
        DateFormat df =
                new SimpleDateFormat("yyyy-MM-dd HH:mm:ss SSS");
        System.out.println("日志记录："+user+"于"+
                df.format(new Date())+"时保存了一条销售记录，客户是"
                    +customer+",购买记录是"+saleModel);
        return f;
    }
}
```

（6）组合使用这些装饰器。

在组合的时候，权限控制应该是最先被执行的，所以把它组合在最外面，日志记录的装饰器会先调用原始的业务对象，所以把日志记录的装饰器组合在中间。

前面讲过，装饰器之间最好不要有顺序限制，但是在实际应用中，可以根据具体的功能要求而有顺序的限制，但应该尽量避免这种情况。

此时客户端测试代码示例如下。

```
public class Client {
    public static void main(String[] args) {
        //得到业务接口，组合装饰器
        GoodsSaleEbi ebi = new CheckDecorator(
                            new LogDecorator(
                                new GoodsSaleEbo()));
        //准备测试数据
        SaleModel saleModel = new SaleModel();
        saleModel.setGoods("Moto 手机");
        saleModel.setSaleNum(2);
        //调用业务功能
        ebi.sale("张三","张三丰", saleModel);
        ebi.sale("李四","张三丰", saleModel);
    }
}
```

运行结果如下：

```
张三保存了张三丰购买 商品名称=Moto 手机,购买数量=2 的销售数据
日志记录：张三于 2010-02-12 16:38:56 730 时保存了一条销售记录，客户是张三丰，
购买记录是商品名称=Moto 手机,购买数量=2
对不起李四，你没有保存销售单的权限
```


好好体会一下，是不是也在没有惊动原始业务对象的情况下，给它织入了新的功能呢？也就是说是在原始业务不知情的情况下，给原始业务对象透明地增加了新功能，从而模拟实现了 AOP 的功能。

事实上，这种做法完全可以应用在项目开发上，在后期为项目的业务对象添加数据检查、权限控制、日志记录等功能，而不需要在业务对象上去处理这些功能了，业务对象可以更专注于具体业务的处理。

22.3.4 装饰模式的优缺点

装饰模式有以下优点。

- 比继承更灵活

  从为对象添加功能的角度来看，装饰模式比继承更灵活。继承是静态的，而且一旦继承所有子类都有一样的功能。而装饰模式采用把功能分离到每个装饰器当中，然后通过对象组合的方式，在运行时动态地组合功能，每个被装饰的对象最终有哪些功能，是由运行期动态组合的功能来决定的。
- 更容易复用功能

  装饰模式把一系列复杂的功能分散到每个装饰器当中，一般一个装饰器只实现一个功能，使实现装饰器变得简单，更重要的是这样有利于装饰器功能的复用，可以给一个对象增加多个同样的装饰器，也可以把一个装饰器用来装饰不同的对象，从而实现复用装饰器的功能。
- 简化高层定义

  装饰模式可以通过组合装饰器的方式，为对象增添任意多的功能。因此在进行高层定义的时候，不用把所有的功能都定义出来，而是定义最基本的就可以了，可以在需要使用的时候，组合相应的装饰器来完成所需的功能。

装饰模式的缺点是：会产生很多细粒度对象。

前面说了，装饰模式是把一系列复杂的功能，分散到每个装饰器当中，一般一个装饰器只实现一个功能，这样会产生很多细粒度的对象，而且功能越复杂，需要的细粒度对象越多。

22.3.5 思考装饰模式

1. 装饰模式的本质

> 装饰模式的本质：**动态组合**。

动态是手段，组合才是目的。这里的组合有两个意思，一个是动态功能的组合，也就是动态进行装饰器的组合；另外一个是指对象组合，通过对象组合来实现为被装饰对象透明地增加功能。

但是要注意，装饰模式不仅可以增加功能，而且也可以控制功能的访问，完全实现新的功能，还可以控制装饰的功能是在被装饰功能之前还是之后来运行等。

总之，装饰模式是通过把复杂功能简单化、分散化，然后在运行期间，根据需要来动态组合的这样一个模式。

2. 何时选用装饰模式

建议在以下情况中选用装饰模式。

- 如果需要在不影响其他对象的情况下，以动态、透明的方式给对象添加职责，可以使用装饰模式，这几乎就是装饰模式的主要功能。
- 如果不适合使用子类来进行扩展的时候，可以考虑使用装饰模式。因为装饰模式是使用的“对象组合”的方式。所谓不适合用子类扩展的方式，比如，扩展功能需要的子类太多，造成子类数目呈爆炸性增长。

22.3.6　相关模式

- 装饰模式与适配器模式

  这是两个没有什么关联的模式，放到一起来说，是因为它们有一个共同的别名：Wrapper。

  这两个模式功能上是不一样的，适配器模式是用来改变接口的，而装饰模式是用来改变对象功能的。
- 装饰模式与组合模式

  这两个模式有相似之处，都涉及到对象的递归调用，从某个角度来说，可以把装饰看做是只有一个组件的组合。

  但是它们的目的完全不一样，装饰模式是要动态地给对象增加功能；而组合模式是想要管理组合对象和叶子对象，为它们提供一个一致的操作接口给客户端，方便客户端的使用。
- 装饰模式与策略模式

  这两个模式可以组合使用。

  策略模式也可以实现动态地改变对象的功能，但是策略模式只是一层选择，也就是根据策略选择一下具体的实现类而已。而装饰模式不是一层，而是递归调用，无数层都可以，只要组合好装饰器的对象组合，那就可以依次调用下去。所以装饰模式更灵活。

  而且策略模式改变的是原始对象的功能，不像装饰模式，后面一个装饰器，改变的是经过前一个装饰器装饰后的对象。也就是策略模式改变的是对象的内核，而装饰模式改变的是对象的外壳。

  这两个模式可以组合使用，可以在一个具体的装饰器中使用策略模式来选择更具体的实现方式。比如前面计算奖金的另外一个问题就是参与计算的基数不同，奖金的计算方式也是不同的。举例来说：假设张三和李四参与同一个奖金的计算，张三的销售总额是 2 万元，而李四的销售总额是 8 万元，它们的计算公式是不一样的，假设奖金的计算规则是，销售额在 5 万以下，统一 3%，而 5 万以上，5 万内是 4%，超过部分是 6%。

  参与同一个奖金的计算，这就意味着可以使用同一个装饰器，但是在装饰器的内部，不同条件下计算公式不一样，那么怎么选择具体的实现策略呢？自然使用策略模式就可以了，也就是装饰模式和策略模式组合来使用。
- 装饰模式与模板方法模式

  这是两个功能上有相似点的模式。

模板方法模式主要应用在算法骨架固定的情况，那么要是算法步骤不固定呢，也就是一个相对动态的算法步骤，就可以使用装饰模式了，因为在使用装饰模式的时候，进行装饰器的组装，其实也相当于是一个调用算法步骤的组装，相当于是一个动态的算法骨架。

既然装饰模式可以实现动态的算法步骤的组装和调用，那么把这些算法步骤固定下来，那就是模板方法模式实现的功能了，因此装饰模式可以模拟实现模板方法模式的功能。

注意 **但是请注意，仅仅只是可以模拟功能而已，两个模式的设计目的、原本的功能、本质思想等都是不一样的。**

第 23 章　职责链模式（Chain of Responsibility）

23.1 场 景 问 题

23.1.1 申请聚餐费用

来考虑这样一个功能：申请聚餐费用的管理。

很多公司都有这样的福利，就是项目组或者是部门可以向公司申请一些聚餐费用，用于组织项目组成员或者是部门成员进行聚餐活动，以增进人员之间的情感，更有利于工作中的相互合作。

申请聚餐费用的大致流程一般是：由申请人先填写申请单，然后交给领导审查，如果申请批准下来了，领导会通知申请人审批通过，然后申请人去财务核领费用，如果没有核准，领导会通知申请人审批未通过，此事也就此作罢。

不同级别的领导，对于审批的额度是不一样的，比如，项目经理只能审批 500 元以内的申请；部门经理能审批 1000 元以内的申请；而总经理可以审核任意额度的申请。

也就是说，当某人提出聚餐费用申请的请求后，该请求会由项目经理、部门经理、总经理之中的某一位领导来进行相应的处理，但是提出申请的人并不知道最终会由谁来处理他的请求，一般申请人是把自己的申请提交给项目经理，或许最后是由总经理来处理他的请求，但是申请人并不知道应该由总经理来处理他的申请请求。

那么该怎样实现这样的功能呢？

23.1.2 不用模式的解决方案

分析上面要实现的功能，主要就是要根据申请费用的多少，然后让不同的领导来进行处理就可以实现了。也就是有点逻辑判断而已。示例代码如下：

```
/**
 * 处理聚餐费用申请的对象
 */
public class FeeRequest {
    /**
     * 提交聚餐费用申请给项目经理
     * @param user 申请人
     * @param fee 申请费用
     * @return 成功或失败的具体通知
     */
    public String requestToProjectManager(String user,double fee){
        String str = "";
        if(fee < 500){
            //项目经理的权限比较小，只能在500以内
            str = this.projectHandle(user, fee);
        }else if(fee < 1000){
```

```
            //部门经理的权限只能在 1000 以内
            str = this.depManagerHandle(user, fee);
        }else if(fee >= 1000){
            //总经理的权限很大，只要请求到了这里，他都可以处理
            str = this.generalManagerHandle(user, fee);
        }
        return str;
    }
    /**
     * 项目经理审批费用申请，参数、返回值和上面是一样的，省略了
     */
    private String projectHandle(String user, double fee) {
        String str = "";
        //为了测试，简单点，只同意小李的
        if("小李".equals(user)){
            str = "项目经理同意"+user+"聚餐费用"+fee+"元的请求";
        }else{
            //其他人一律不同意
            str = "项目经理不同意"+user+"聚餐费用"+fee+"元的请求";
        }
        return str;
    }
    /**
     * 部门经理审批费用申请，参数、返回值和上面是一样的，省略了
     */
    private String depManagerHandle(String user, double fee) {
        String str = "";
        //为了测试，简单点，只同意小李申请的
        if("小李".equals(user)){
            str = "部门经理同意"+user+"聚餐费用"+fee+"元的请求";
        }else{
            //其他人一律不同意
            str= "部门经理不同意"+user+"聚餐费用"+fee+"元的请求";
        }
        return str;
    }
    /**
     * 总经理审批费用申请，参数、返回值和上面是一样的，省略了
     */
    private String generalManagerHandle(String user, double fee) {
```

```
        String str = "";

        //为了测试，简单点，只同意小李的
        if("小李".equals(user)){
            str = "总经理同意"+user+"聚餐费用"+fee+"元的请求";
        }else{
            //其他人一律不同意
            str = "总经理不同意"+user+"聚餐费用"+fee+"元的请求";
        }
        return str;
    }
}
```

写个客户端来测试看看效果。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        FeeRequest request = new FeeRequest();

        //开始测试
        String ret1 = request.requestToProjectManager("小李", 300);
        System.out.println("the ret="+ret1);
        String ret2 = request.requestToProjectManager("小张", 300);
        System.out.println("the ret="+ret2);

        String ret3 = request.requestToProjectManager("小李", 600);
        System.out.println("the ret="+ret3);
        String ret4 = request.requestToProjectManager("小张", 600);
        System.out.println("the ret="+ret4);

        String ret5 = request.requestToProjectManager("小李", 1200);
        System.out.println("the ret="+ret5);
        String ret6 = request.requestToProjectManager("小张", 1200);
        System.out.println("the ret="+ret6);
    }
}
```

运行结果如下：

```
the ret1=项目经理同意小李聚餐费用 300.0 元的请求
the ret2=项目经理不同意小张聚餐费用 300.0 元的请求
the ret3=部门经理同意小李聚餐费用 600.0 元的请求
the ret4=部门经理不同意小张聚餐费用 600.0 元的请求
the ret5=总经理同意小李聚餐费用 1200.0 元的请求
```

```
the ret6=总经理不同意小张聚餐费用 1200.0 元的请求
```

23.1.3　有何问题

上面的实现很简单，基本上没有什么特别的难度。仔细想想，这么实现有没有问题呢？仔细分析申请聚餐费用的业务功能和目前的实现，主要面临着以下问题。

- 聚餐费用申请的处理流程是可能会变动的。

  比如现在的处理流程是：提交申请给项目经理，看看是否适合由项目经理处理，如果不是→看看是否适合由部门经理处理，如果不是→总经理处理的步骤。今后可能会变化成：直接提交给部门经理，看看是否适合由部门经理处理，如果不是→总经理处理这样的步骤。也就是说，对于聚餐费用申请，要求处理的逻辑步骤是灵活的。

- 各个处理环节的业务处理也是会变动的。

  因为处理流程可能发生变化，也会导致某些步骤具体的业务功能发生变化，比如，原本部门经理审批聚餐费用的时候，只是判断是否批准；现在，部门经理可能在审批聚餐费用的时候，核算本部门的实时成本，这就出现新的业务处理功能了。

采用上面的实现，如果处理的逻辑发生了变化，解决的方法，一个是生成一个子类，覆盖 requestToProjectManager 方法，然后在里面实现新的处理；另外一个方法就是修改处理申请方法的源代码来实现。要是具体处理环节的业务处理功能发生了变化，那就只好找到相应的处理方法，进行源代码修改了。

总之都不是什么好方法，也就是说，如果出现聚餐费用申请的处理流程变化的情况，或者是出现各个处理环节的功能变化的时候，上面的实现方式是很难灵活地变化来适应新功能的要求的。

把上面的问题抽象一下：客户端发出一个请求，会有很多对象都可以来处理这个请求，而且不同对象的处理逻辑是不一样的。对于客户端而言，无所谓谁来处理，反正有对象处理就可以了。而且在上述处理中，还希望处理流程是可以灵活变动的，而处理请求的对象需要能方便地修改或者是被替换掉，以适应新的业务功能的需要。

请问如何才能实现上述要求？

23.2　解 决 方 案

23.2.1　使用职责链模式来解决问题

用来解决上述问题的一个合理的解决方案，就是使用职责链模式。那么什么是职责链模式呢？

1. 职责链模式的定义

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

2. 应用职责链模式来解决问题的思路

仔细分析上面的场景，当客户端提出一个聚餐费用的申请，后续处理这个申请的对象项目经理、部门经理和总经理，自然地形成了一个链，从项目经理→部门经理→总经理，客户端的申请请求就在这个链中传递，直到有领导处理为止。看起来，上面的功能要求很适合采用职责链来处理这个业务。

要想让处理请求的流程可以灵活地变动，**一个基本的思路，那就是动态构建流程步骤，这样随时都可以重新组合出新的流程来**。而要让处理请求的对象也要很灵活，那就要让它足够简单，最好是只实现单一的功能，或者是有限的功能，这样更有利于修改和复用。

职责链模式就很好地体现了上述的基本思路，首先职责链模式会定义一个所有处理请求的对象都要继承实现的抽象类，这样就有利于随时切换新的实现；其次每个处理请求对象只实现业务流程中的一步业务处理，这样使其变得简单；最后职责链模式会动态地来组合这些处理请求的对象，把它们按照流程动态地组合起来，并要求它们依次调用，这样就动态地实现了流程。

这样一来，如果流程发生了变化，只要重新组合就可以了；如果某个处理的业务功能发生了变化，一个方案是修改该处理对应的处理对象；另一个方案是直接提供一个新的实现，然后在组合流程的时候，用新的实现替换掉旧的实现就可以了。

23.2.2 职责链模式的结构和说明

职责链模式的结构如图 23.1 所示。

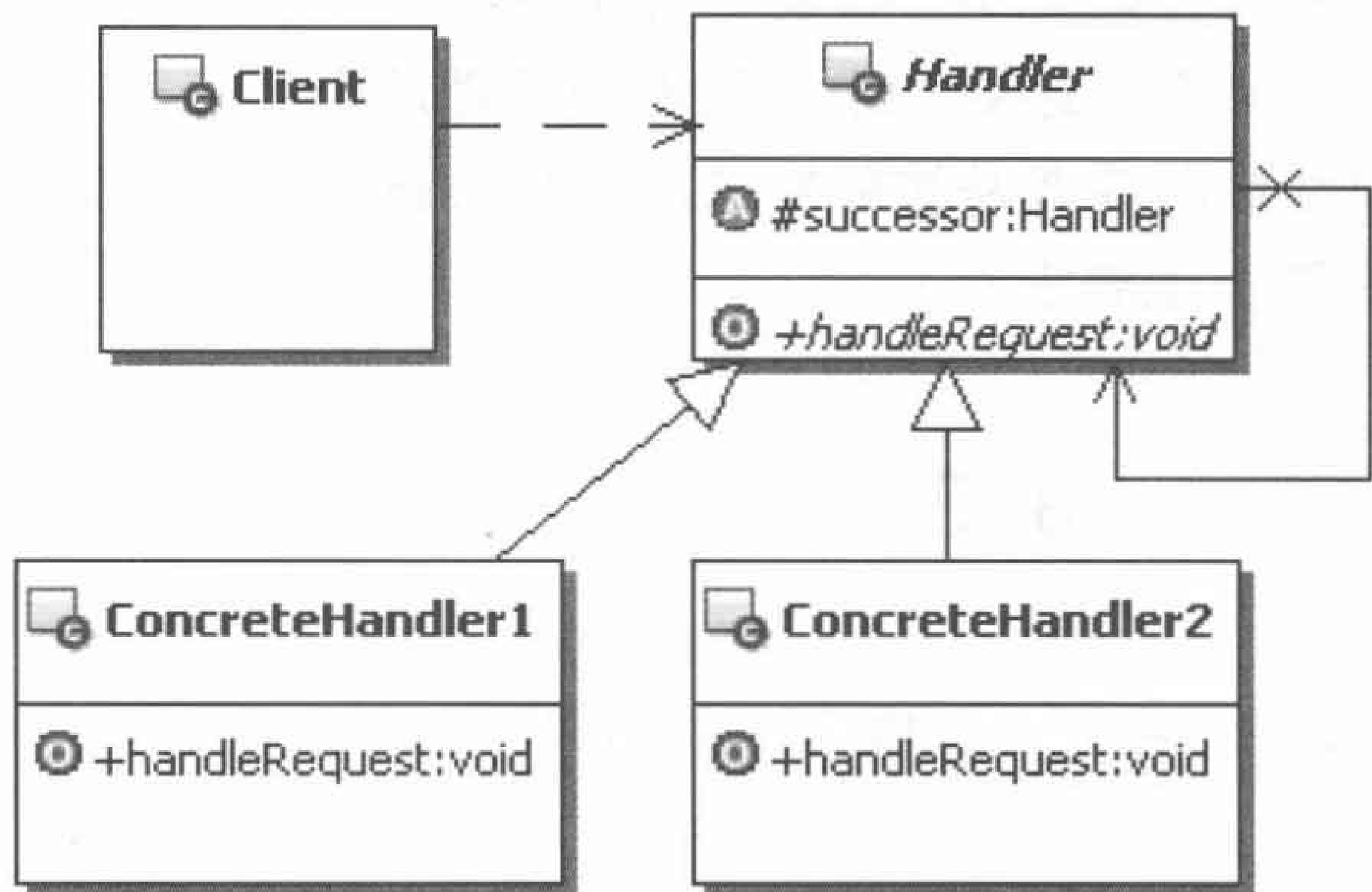


图 23.1 职责链模式结构图

- Handler：定义职责的接口，通常在这里定义处理请求的方法，可以在这里实现后继链。
- ConcreteHandler：实现职责的类，在这个类中，实现对在它职责范围内请求的

处理，如果不处理，就继续转发请求给后继者。

- Client：职责链的客户端，向链上的具体处理对象提交请求，让职责链负责处理。

23.2.3　职责链模式示例代码

（1）先来看看职责的接口定义。示例代码如下：

```
/**
 * 职责的接口，也就是处理请求的接口
 */
public abstract class Handler {
    /**
     * 持有后继的职责对象
     */
    protected Handler successor;
    /**
     * 设置后继的职责对象
     * @param successor 后继的职责对象
     */
    public void setSuccessor(Handler successor) {
        this.successor = successor;
    }
    /**
     * 示意处理请求的方法，虽然这个示意方法是没有传入参数的
     * 但实际是可以传入参数的，根据具体需要来选择是否传递参数
     */
    public abstract void handleRequest();
}
```

（2）再来看看具体的职责实现对象。示例代码如下：

```
/**
 * 具体的职责对象，用来处理请求
 */
public class ConcreteHandler1 extends Handler {
    public void handleRequest() {
        //根据某些条件来判断是否属于自己处理的职责范围
        //判断条件比如，从外部传入的参数，或者这里主动去获取的外部数据，
        //如从数据库中获取等，下面这句话只是个示意
        boolean someCondition = false;

        if(someCondition){
            //如果属于自己处理的职责范围，就在这里处理请求
```

```
                //具体的处理代码
                System.out.println("ConcreteHandler1 handle request");
            }else{
                //如果不属于自己处理的职责范围，那就判断是否还有后继的职责对象
                //如果有，就转发请求给后继的职责对象
                //如果没有，什么都不做，自然结束
                if(this.successor!=null){
                    this.successor.handleRequest();
                }
            }
        }
}
```

另外，ConcreteHandler2 和 ConcreteHandler1 的示意代码几乎是一样的，因此就不再赘述。

（3）接下来看看客户端的示意。示例代码如下：

```
/**
 * 职责链的客户端，这里只是个示意
 */
public class Client {
    public static void main(String[] args) {
        //先要组装职责链
        Handler h1 = new ConcreteHandler1();
        Handler h2 = new ConcreteHandler2();

        h1.setSuccessor(h2);
        //然后提交请求
        h1.handleRequest();
    }
}
```

23.2.4 使用职责链模式重写示例

要使用职责链模式来重写示例，先来实现如下的功能：当某人提出聚餐费用申请的请求后，该请求会在项目经理→部门经理→总经理这样一条领导处理链上进行传递，发出请求的人并不知道谁会来处理他的请求，每个领导会根据自己的职责范围，来判断是处理请求还是把请求交给更高级的领导，只要有领导处理了，传递就结束了。

需要把每位领导的处理独立出来，实现成单独的职责处理对象，然后为它们提供一个公共的、抽象的父职责对象，这样就可以在客户端来动态地组合职责链，实现不同的功能要求了。还是看一下示例的整体结构，有助于对示例的理解和把握，如图 23.2 所示。

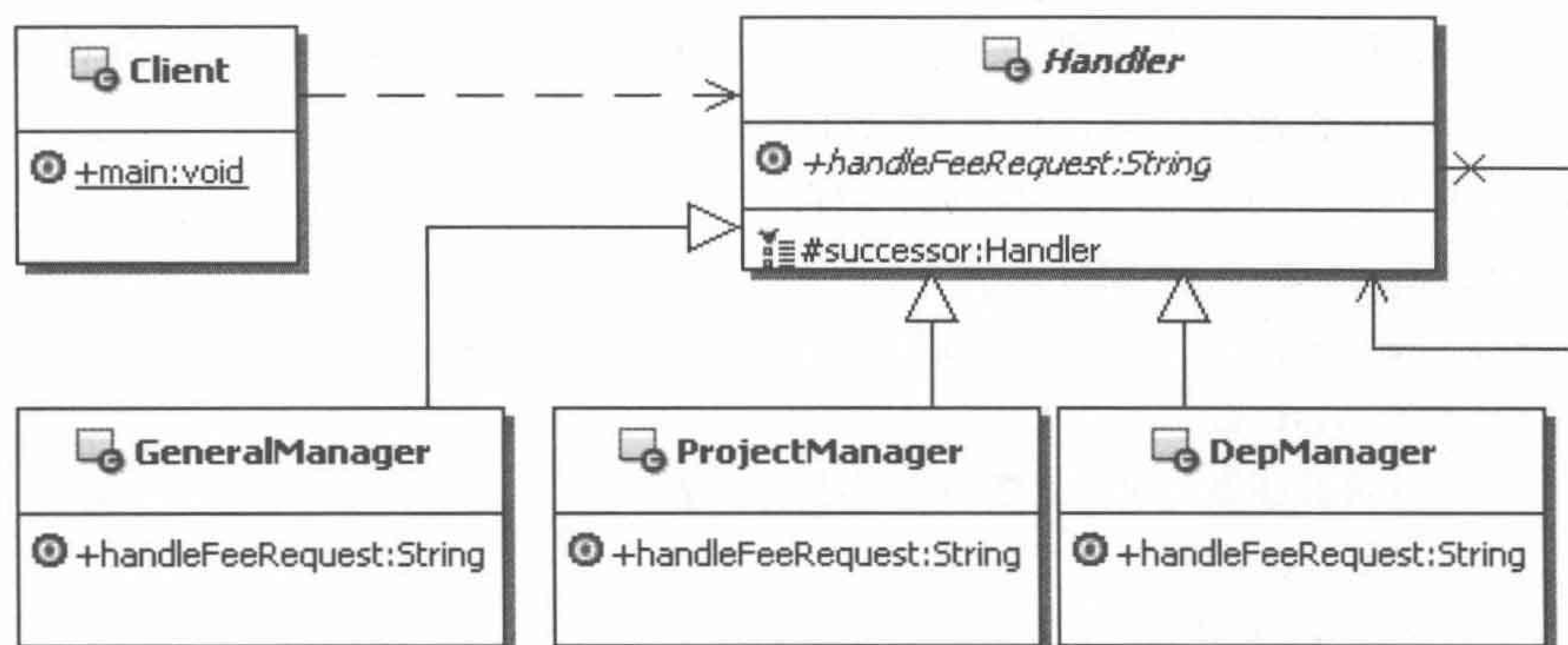


图 23.2　使用职责链模式的示例程序的结构示意图

1. 定义职责的抽象类

首先来看看定义所有职责的抽象类，也就是所有职责的外观，在这个类中持有下一个处理请求的对象，同时还要定义业务处理方法。示例代码如下：

```
/**
 * 定义职责对象的接口
 */
public abstract class Handler {
    /**
     * 持有下一个处理请求的对象
     */
    protected Handler successor = null;
    /**
     * 设置下一个处理请求的对象
     * @param successor 下一个处理请求的对象
     */
    public void setSuccessor(Handler successor){
        this.successor = successor;
    }
    /**
     * 处理聚餐费用的申请
     * @param user 申请人
     * @param fee 申请的钱数                这是个业务方法
     * @return 成功或失败的具体通知
     */
    public abstract String handleFeeRequest(String user,double fee);
}
```

2. 实现各自的职责

现在实现的处理聚餐费用流程是：申请人提出的申请交给项目经理处理，项目经理的处理权限是 500 元以内，超过 500 元，把申请转给部门经理处理，部门经理的处理权限是 1000 元以内，超过 1000 元，把申请转给总经理处理。

分析上述流程，该请求主要有三个处理环节，把它们分别实现成为职责对象，一个对象实现一个环节的处理功能，这样就会比较简单。

先看看项目经理的处理吧。示例代码如下：

```
public class ProjectManager extends Handler{
    public String handleFeeRequest(String user, double fee) {
        String str = "";
        //项目经理的权限比较小，只能在 500 以内
        if(fee < 500){
            //为了测试，简单点，只同意小李的
            if("小李".equals(user)){
                str = "项目经理同意"+user+"聚餐费用"+fee+"元的请求";
            }else{
                //其他人一律不同意
                str = "项目经理不同意"+user+"聚餐费用"+fee+"元的请求";
            }
            return str;
        }else{
            //超过 500，继续传递给级别更高的人处理
            if(this.successor!=null){
                return successor.handleFeeRequest(user, fee);
            }
        }
        return str;
    }
}
```

接下来看看部门经理的处理。示例代码如下：

```
public class DepManager extends Handler{
    public String handleFeeRequest(String user, double fee) {
        String str = "";
        //部门经理的权限只能在 1000 以内
        if(fee < 1000){
            //为了测试，简单点，只同意小李申请的
            if("小李".equals(user)){
                str = "部门经理同意"+user+"聚餐费用"+fee+"元的请求";
            }else{
                //其他人一律不同意
                str = "部门经理不同意"+user+"聚餐费用"+fee+"元的请求";
            }
            return str;
        }else{
```

```
            //超过 1000，继续传递给级别更高的人处理
            if(this.successor!=null){
                return this.successor.handleFeeRequest(user, fee);
            }
        }
        return str;
    }
}
```

再看总经理的处理。示例代码如下：

```
public class GeneralManager extends Handler{
    public String handleFeeRequest(String user, double fee) {
        String str = "";
        //总经理的权限很大，只要请求到了这里，他都可以处理
        if(fee >= 1000){
            //为了测试，简单点，只同意小李的
            if("小李".equals(user)){
                str = "总经理同意"+user+"聚餐费用"+fee+"元的请求";
            }else{
                //其他人一律不同意
                str = "总经理不同意"+user+"聚餐费用"+fee+"元的请求";
            }
            return str;
        }else{
            //如果还有后继的处理对象，继续传递
            if(this.successor!=null){
                return successor.handleFeeRequest(user, fee);
            }
        }
        return str;
    }
}
```

3. 使用职责链

那么客户端如何使用职责链呢，最重要的就是要先构建职责链，然后才能使用。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //先要组装职责链
        Handler h1 = new GeneralManager();
        Handler h2 = new DepManager();
        Handler h3 = new ProjectManager();
```

```
        h3.setSuccessor(h2);
        h2.setSuccessor(h1);

        //开始测试
        String ret1 = h3.handleFeeRequest("小李", 300);
        System.out.println("the ret1="+ret1);
        String ret2 = h3.handleFeeRequest("小张", 300);
        System.out.println("the ret2="+ret2);

        String ret3 = h3.handleFeeRequest("小李", 600);
        System.out.println("the ret3="+ret3);
        String ret4 = h3.handleFeeRequest("小张", 600);
        System.out.println("the ret4="+ret4);

        String ret5 = h3.handleFeeRequest("小李", 1200);
        System.out.println("the ret5="+ret5);
        String ret6 = h3.handleFeeRequest("小张", 1200);
        System.out.println("the ret6="+ret6);
    }
}
```

运行结果如下：

```
the ret1=项目经理同意小李聚餐费用 300.0 元的请求
the ret2=项目经理不同意小张聚餐费用 300.0 元的请求
the ret3=部门经理同意小李聚餐费用 600.0 元的请求
the ret4=部门经理不同意小张聚餐费用 600.0 元的请求
the ret5=总经理同意小李聚餐费用 1200.0 元的请求
the ret6=总经理不同意小张聚餐费用 1200.0 元的请求
```

看起来结果跟前面不用模式的实现方案的运行结果是一样的，它们本来就是实现的同样的功能，只不过实现方式不同而已。

4. 如何运行的

理解了示例的整体结构和具体实现，那么示例的具体运行过程是怎样的呢？

下面就以“小李申请聚餐费用 1200 元”这个费用申请为例来说明。调用过程的示意图如图 23.3 所示。

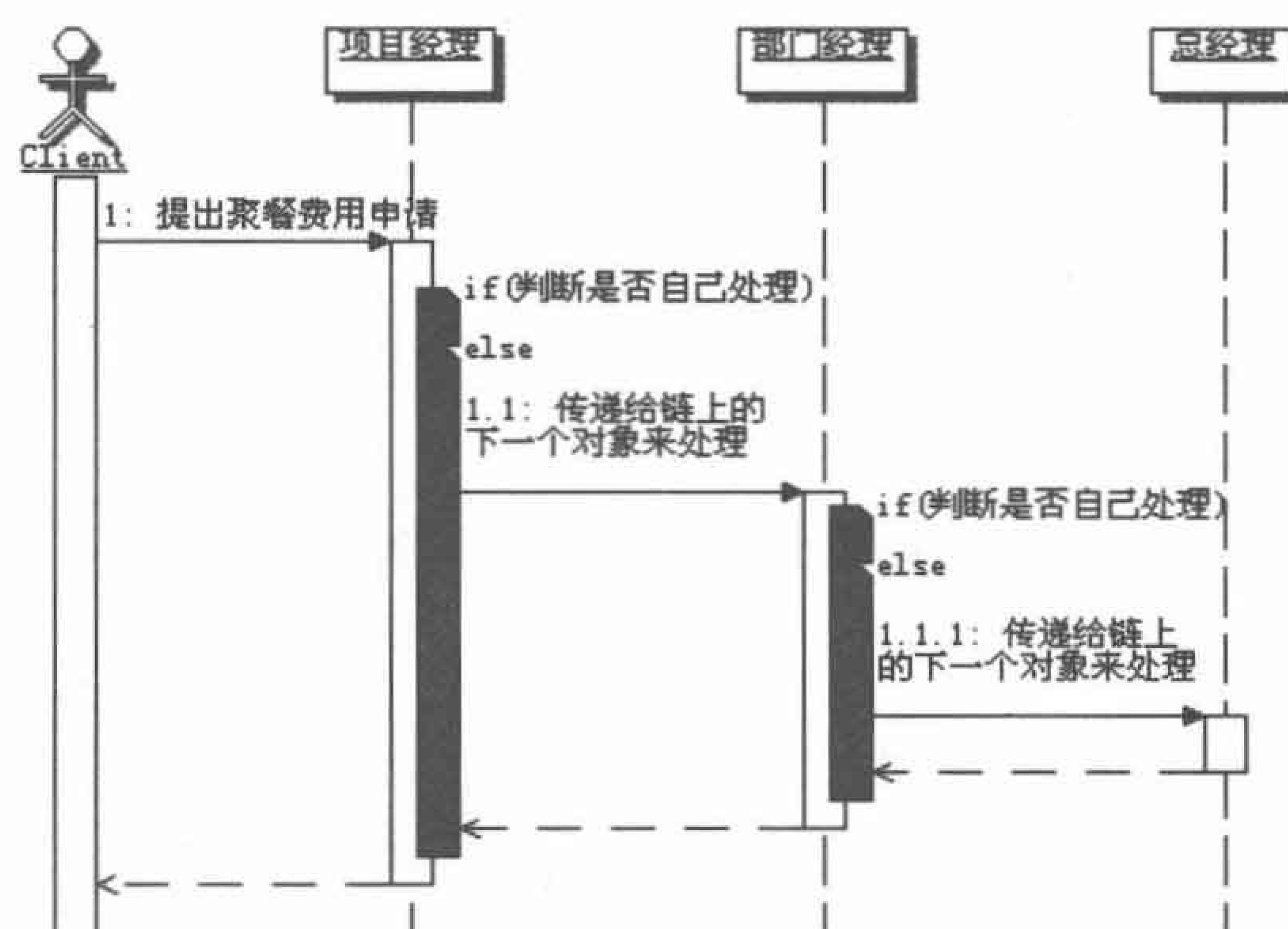


图 23.3　职责链示例调用过程示意图

23.3 模 式 讲 解

23.3.1 认识职责链模式

1. 职责链模式的功能

职责链模式主要用来处理“客户端发出一个请求，有多个对象都有机会来处理这一个请求，但是客户端不知道究竟谁会来处理他的请求”这样的情况。也就是需要让请求者和接收者解耦，这样就可以动态地切换和组合接收者了。

> 注意：**要注意在标准的职责链模式中，只要有对象处理了请求，这个请求就到此为止，不再被传递和处理了。**

如果要变形使用职责链，就可以让这个请求继续传递，每个职责对象对这个请求进行一定的功能处理，从而形成一个处理请求的功能链。

2. 隐式接收者

当客户端发出请求的时候，客户端并不知道谁会真正处理他的请求，客户端只知道他提交请求的第一个对象。从第一个处理对象开始，整个职责链中的对象，要么自己处理请求，要么继续转发给下一个接收者。

也就是对于请求者而言，并不知道最终的接收者是谁，但是一般情况下，总是会有一个对象来处理的，因此称为**隐式接收者**。

3. 如何构建链

职责链的链怎么构建呢？这是个大问题，实现的方式也是五花八门，归结起来大致有以下一些方式。

首先是按照实现的地方来说：

- 可以实现在客户端提交请求前组合链。也就是在使用的时候动态组合链，称为外部链；
- 也可以在 Handler 里面实现链的组合，算是内部链的一种；
- 当然还有一种就是在各个职责对象中，由各个职责对象自行决定后续的处理对象。这种实现方式要求每个职责对象除了进行业务处理外，还必须了解整个业务流程。

按照构建链的数据来源，也就是决定了按照什么顺序来组合链的数据，又分为以下几种。

- 一种就是在程序中动态组合。
- 也可以通过外部，比如数据库来获取组合的数据，这种属于数据库驱动的方式。
- 还有一种方式就是通过配置文件传递进来，也可以是流程的配置文件。

如果是从外部获取数据来构建链，那么在程序运行的时候，会读取这些数据，然后根据数据的要求来获取相应的对象，并组合起来。

还有一种是不需要构建链，因为已有的对象已经自然构成链了，这种情况多出现在组合模式构建的对象树中，这样子对象可以很自然地向上找到自己的父对象。就像部门

人员的组织结构一样，顶层是总经理，总经理下面是各个部门的经理，部门经理下面是项目经理，项目经理下面是各个普通员工，自然就可以形成：普通员工→项目经理→部门经理→总经理这样的链。

4. 谁来处理

职责链中那么多处理对象，到底谁来处理请求呢，这个是在运行时期动态决定的。当请求被传递到某个处理对象的时候，这个对象会按照已经设定好的条件来判断是否属于自己处理的范围，如果是就处理，如果不是就转发请求给下一个对象。

5. 请求一定会被处理吗

> 提示　**在职责链模式中，请求不一定会被处理，因为可能没有合适的处理者，请求在职责链中从头传递到尾，每个处理对象都判断不属于自己处理，最后请求就没有对象来处理。这一点是需要注意的。**

可以在职责链的末端始终加上一个不支持此功能处理的职责对象，这样如果传递到这里，就会出现提示，本职责链没有对象处理这个请求。

23.3.2　处理多种请求

前面的示例都是同一个职责链处理一种请求的情况，现在有这样的需求，还是费用申请的功能，这次是申请预支差旅费，假设还是同一流程，也就是组合同一个职责链，从项目经理→传递给部门经理→传递给总经理，虽然流程相同，但是每个处理类需要处理两种请求，它们的具体业务逻辑是不一样的，那么该如何实现呢？

1. 简单的处理方式

要解决这个问题，也不是很困难，一个简单的方法就是为每种业务单独定义一个方法，然后客户端根据不同的需要调用不同的方法。还是通过代码来示例一下。注意这里故意把两个方法做得有些不一样，一个是返回 String 类型的值，一个是返回 boolean 类型的值；另外一个是返回到客户端再输出信息，一个是直接在职责处理中就输出信息。

（1）首先是改造职责对象的接口，添加上新的业务方法。示例代码如下：

```
/**
 * 定义职责对象的接口
 */
public abstract class Handler {
    /**
     * 持有下一个处理请求的对象
     */
    protected Handler successor = null;
    /**
     * 设置下一个处理请求的对象
     * @param successor 下一个处理请求的对象
     */
```

```
    public void setSuccessor(Handler successor){
        this.successor = successor;
    }
    /**
     * 处理聚餐费用的申请
     * @param user 申请人
     * @param fee 申请的钱数
     * @return 成功或失败的具体通知
     */
    public abstract String handleFeeRequest(String user,double fee);
    /**
     * 处理预支差旅费用的申请
     * @param user 申请人
     * @param requestFee 申请的钱数
     * @return 是否同意
     */
    public abstract boolean handlePreFeeRequest(
                                        String user,double requestFee);
}
```

新加的业务处理方法

（2）职责的接口发生了改变，对应的处理类也要改变，这几个处理类是类似的，原有的功能不变，然后在新的实现方法中，同样判断一下是否属于自己处理的范围，如果属于自己处理的范围那就处理，否则就传递到下一个处理。还是示范一个，看看项目经理的处理吧。示例代码如下：

```
public class ProjectManager extends Handler{
    public String handleFeeRequest(String user, double fee) {
        String str = "";
        //项目经理的权限比较小，只能在 500 以内
        if(fee < 500){
            //为了测试，简单点，只同意小李的
            if("小李".equals(user)){
                str = "项目经理同意"+user+"聚餐费用"+fee+"元的请求";
            }else{
                //其他人一律不同意
                str = "项目经理不同意"+user+"聚餐费用"+fee+"元的请求";
            }
            return str;
        }else{
            //超过 500，继续传递给级别更高的人处理
            if(this.successor!=null){
                return successor.handleFeeRequest(user, fee);
```

这个方法的处理没有任何变化

```
            }
        }
        return str;
    }
    public boolean handlePreFeeRequest(String user, double
requestNum) {
        //项目经理的权限比较小，只能在 5000 以内
        if(requestNum < 5000){
            //工作需要嘛，统统同意
            System.out.println("项目经理同意"+user
                        +"预支差旅费用"+requestNum+"元的请求");
            return true;
        }else{
            //超过 5000，继续传递给级别更高的人处理
            if(this.successor!=null){
                return this.successor.handlePreFeeRequest(
                                        user, requestNum);
            }
        }
        return false;
    }
}
```

新加的业务处理方法

其他的处理类似，就不在演示了。

（3）准备好了各个处理职责的类，看看客户端如何调用。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //先要组装职责链
        Handler h1 = new GeneralManager();
        Handler h2 = new DepManager();
        Handler h3 = new ProjectManager();
        h3.setSuccessor(h2);
        h2.setSuccessor(h1);

        //开始测试申请聚餐费用
        String ret1 = h3.handleFeeRequest("小李", 300);
        System.out.println("the ret1="+ret1);
        String ret2 = h3.handleFeeRequest("小李", 600);
        System.out.println("the ret2="+ret2);
        String ret3 = h3.handleFeeRequest("小李", 1200);
        System.out.println("the ret3="+ret3);
```

组装职责链的步骤是不变的，下面的测试也是在同一个职责链上测试

```
        //开始测试申请差旅费用
        h3.handlePreFeeRequest("小张", 3000);
        h3.handlePreFeeRequest("小张", 6000);
        h3.handlePreFeeRequest("小张", 32000);
    }
}
```


运行的结果如下：

```
the ret1=项目经理同意小李聚餐费用 300.0 元的请求
the ret2=部门经理同意小李聚餐费用 600.0 元的请求
the ret3=总经理同意小李聚餐费用 1200.0 元的请求
项目经理同意小张预支差旅费用 3000.0 元的请求
部门经理同意小张预支差旅费用 6000.0 元的请求
总经理同意小张预支差旅费用 32000.0 元的请求
```

2. 通用请求的处理方式

上面的实现看起来很容易，但是仔细想想，这样实现有没有什么问题呢？

这种实现方式有一个很明显的问题，那就是只要增加一个业务，就需要修改职责的接口，这是很不灵活的，Java 开发中很强调面向接口编程，因此接口应该相对保持稳定，接口一改，需要修改的地方就太多了，频繁修改接口绝对不是个好办法。

那有没有什么好方法来实现呢？分析一下现在变化的东西。

- 一是不同的业务需要传递的业务数据不同；
- 二是不同的业务请求的方法不同；
- 三是不同的职责对象处理这些不同的业务请求的业务逻辑不同。

现在有一种简单的方式，可以较好地解决这些问题。首先定义一套通用的调用框架，用一个通用的请求对象来封装请求传递的参数；然后定义一个通用的调用方法，这个方法不去区分具体业务，所有的业务都是这一个方法，那么具体的业务如何区分呢？就是在通用的请求对象中会有一个业务的标记；到了职责对象中，愿意处理就和原来使用一样的处理方式，如果不愿意处理，则传递到下一个处理对象就可以了。

对于返回值也可以来个通用的，最简单的是使用 Object 类型。

看例子吧，为了示范，先假定只有一个业务方法，等把这一个方法搞定了，明白了，然后再扩展一个业务方法，就能清晰地看出这种设计的好处了。

（1）先看看通用的请求对象的定义。示例代码如下：

```
/**
 * 通用的请求对象
 */
public class RequestModel {
    /**
     * 表示具体的业务类型
     */
    private String type;
```

```
    /**
     * 通过构造方法把具体的业务类型传递进来
     * @param type 具体的业务类型
     */
    public RequestModel(String type){
        this.type = type;
    }
    public String getType() {
        return type;
    }
}
```

（2）看看此时的通用职责处理对象，在这里要实现一个通用的调用框架。示例代码如下：

```
/**
 * 定义职责对象的接口
 */
public abstract class Handler {
    /**
     * 持有下一个处理请求的对象
     */
    protected Handler successor = null;
    /**
     * 设置下一个处理请求的对象
     * @param successor 下一个处理请求的对象
     */
    public void setSuccessor(Handler successor){
        this.successor = successor;
    }
    /**
     * 通用的请求处理方法
     * @param rm 通用的请求对象
     * @return 处理后需要返回的对象
     */
    public Object handleRequest(RequestModel rm){
        if(successor != null){
            //这个是默认的实现，如果子类不愿意处理这个请求
            //那就传递到下一个职责对象去处理
            return this.successor.handleRequest(rm);
        }else{
            System.out.println(
```

```
                        "没有后续处理或者暂时不支持这样的功能处理");
            return false;
        }
    }
}
```

（3）现在来加上第一个业务，就是“聚餐费用申请”的处理，为了描述具体的业务数据，需要扩展通用的请求对象，把业务数据封装进去，另外定义一个请求对象。示例代码如下：

```
/**
 * 封装和聚餐费用申请业务相关的请求数据
 */
public class FeeRequestModel extends RequestModel{
    /**
     * 约定具体的业务类型
     */
    public final static String FEE_TYPE = "fee";

    public FeeRequestModel() {
        super(FEE_TYPE);
    }
    /**
     * 申请人
     */
    private String user;
    /**
     * 申请金额
     */
    private double fee;
    public String getUser() {
        return user;
    }
    public void setUser(String user) {
        this.user = user;
    }
    public double getFee() {
        return fee;
    }
    public void setFee(double fee) {
        this.fee = fee;
    }
```

调用父类的构造方法，约定好业务类型

相应的 getter/setter

```
}
```

（4）接下来该实现职责对象的处理了。下面看看项目经理的处理吧。在这个处理类中，首先要覆盖父类的通用业务处理方法，然后在其中处理自己想要实现的业务，不想处理的就让父类去处理，父类会默认传递给下一个处理对象。示例代码如下：

```
/**
 * 实现项目经理处理聚餐费用申请的对象
 */
public class ProjectManager extends Handler{
    public Object handleRequest(RequestModel rm){
        if(FeeRequestModel.FEE_TYPE
                .equals(rm.getType())){
            //表示聚餐费用申请
            return handleFeeRequest(rm);
        }else{
            //其他的项目经理暂时不想处理
            return super.handleRequest(rm);
        }
    }
    private Object handleFeeRequest(RequestModel rm) {
        //先把通用的对象造型回来
        FeeRequestModel frm =
                (FeeRequestModel)rm;
        String str = "";
        //项目经理的权限比较小，只能在500以内
        if(frm.getFee() < 500){
            //为了测试，简单点，只同意小李的
            if("小李".equals(frm.getUser())){
                str = "项目经理同意"+frm.getUser()
                        +"聚餐费用"+frm.getFee()+"元的请求";
            }else{
                //其他人一律不同意
                str = "项目经理不同意"+frm.getUser()
                        +"聚餐费用"+frm.getFee()+"元的请求";
            }
            return str;
        }else{
```

```
            //超过 500，继续传递给级别更高的人处理
            if(this.successor!=null){
                return successor.handleRequest(rm);
            }
        }
        return str;
    }
}
```

部门经理、总经理的处理对象和项目经理的处理类似，就不再示例了。

（5）客户端也需要变化。对于客户端，唯一的麻烦是需要知道每个业务对应的具体的请求对象，因为要封装业务数据进去。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //先要组装职责链
        Handler h1 = new GeneralManager();
        Handler h2 = new DepManager();
        Handler h3 = new ProjectManager();
        h3.setSuccessor(h2);
        h2.setSuccessor(h1);

        //开始测试申请聚餐费用
        FeeRequestModel frm = new FeeRequestModel();
        frm.setFee(300);
        frm.setUser("小李");
        //调用处理
        String ret1 = (String)h3.handleRequest(frm);
        System.out.println("ret1="+ret1);

        //重新设置申请金额，再调用处理
        frm.setFee(800);
        h3.handleRequest(frm);
        String ret2 = (String)h3.handleRequest(frm);
        System.out.println("ret2="+ret2);

        //重新设置申请金额，再调用处理
        frm.setFee(1600);
        h3.handleRequest(frm);
        String ret3 = (String)h3.handleRequest(frm);
        System.out.println("ret3="+ret3);
    }
```

组装职责链的过程和以前是一样的

```
}
```

运行结果如下：

```
ret1=项目经理同意小李聚餐费用 300.0 元的请求
ret2=部门经理同意小李聚餐费用 800.0 元的请求
ret3=总经理同意小李聚餐费用 1600.0 元的请求
```

（6）接下来看看如何在不改动现有框架的前提下，扩展新的业务，这样才能说明这种设计的灵活性。

假如就是要实现上面示例过的另外一个功能“预支差旅费申请”吧。要想扩展新的业务，第一步就是新建一个封装业务数据的对象。示例代码如下：

```
/**
 * 封装跟预支差旅费申请业务相关的请求数据
 */
public class PreFeeRequestModel extends RequestModel{
    /**
     * 约定具体的业务类型
     */
    public final static String FEE_TYPE = "preFee";
    public PreFeeRequestModel() {
        super(FEE_TYPE);
    }
    /**
     * 申请人
     */
    private String user;
    /**
     * 申请金额
     */
    private double fee;
    public String getUser() {
        return user;
    }
    public void setUser(String user) {
        this.user = user;
    }
    public double getFee() {
        return fee;
    }
    public void setFee(double fee) {
        this.fee = fee;
    }
```

```
}
```

有些朋友会发现，这个对象和封装聚餐费用申请业务数据的对象几乎完全一样。这里要说明一下，一样的原因主要是为了演示简单，设计得相似，实际业务中可能是不一样的，因此，最好还是一个业务一个对象，如果确实有公共的数据，可以定义公共的父类，最好不要让不同的业务使用统一个对象，容易混淆。

（7）对于具体进行职责处理的类，比较好的方式就是扩展出子类来，然后在子类中实现新加入的业务，当然也可以直接在原来的对象上改。下面采用扩展出子类的方式，来看看新的项目经理的处理类。示例代码如下：

```
/**
 * 实现为项目经理增加预支差旅费用申请处理功能的子对象
 * 现在的项目经理既可以处理聚餐费用申请，又可以处理预支差旅费用申请
 */
public class ProjectManager2 extends ProjectManager{
    public Object handleRequest(RequestModel rm){
        if(PreFeeRequestModel.FEE_TYPE.equals(rm.getType())){
            //表示预支差旅费用申请
            return myHandler(rm);
        }else{
            //其他的让父类去处理
            return super.handleRequest(rm);
        }
    }
    private Object myHandler(RequestModel rm) {
        //先把通用的对象造型回来
        PreFeeRequestModel frm = (PreFeeRequestModel)rm;
        //项目经理的权限比较小，只能在 5000 以内
        if(frm.getFee() < 5000){
            //工作需要嘛，全部同意
            System.out.println("项目经理同意"+frm.getUser()
                    +"预支差旅费用"+frm.getFee()+"元的请求");
            return true;
        }else{
            //超过 5000，继续传递给级别更高的人处理
            if(this.successor!=null){
                return this.successor.handleRequest(rm);
            }
        }
        return false;
    }
}
```

故意定义一个和以前不一样的名称。这里可以是任意合法的名称

部门经理和总经理的处理类似于项目经理的处理，这里就不再示例了。

（8）看看此时的测试。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //先要组装职责链
```

```
        Handler h1 = new GeneralManager2();
        Handler h2 = new DepManager2();
        Handler h3 = new ProjectManager2();
        h3.setSuccessor(h2);
        h2.setSuccessor(h1);

        //开始测试申请聚餐费用
        FeeRequestModel frm = new FeeRequestModel();
        frm.setFee(300);
        frm.setUser("小李");
        //调用处理
        String ret1 = (String)h3.handleRequest(frm);
        System.out.println("ret1="+ret1);

        //重新设置申请金额，再调用处理
        frm.setFee(800);
        h3.handleRequest(frm);
        String ret2 = (String)h3.handleRequest(frm);
        System.out.println("ret2="+ret2);

        //重新设置申请金额，再调用处理
        frm.setFee(1600);
        h3.handleRequest(frm);
        String ret3 = (String)h3.handleRequest(frm);
        System.out.println("ret3="+ret3);

        //开始测试申请预支差旅费用
        PreFeeRequestModel pfrm =
                  new PreFeeRequestModel();
        pfrm.setFee(3000);
        pfrm.setUser("小张");
        //调用处理
        h3.handleRequest(pfrm);
        //重新设置申请金额，再调用处理
        pfrm.setFee(6000);
        h3.handleRequest(pfrm);
        //重新设置申请金额，再调用处理
        pfrm.setFee(36000);
        h3.handleRequest(pfrm);
}
```

> 注意这里创建的都是扩展过后的对象，可以同时支持两种业务

> 注意这里测试不同的业务，但调用处理请求的方法是一样的，是通用的方法

```
}
```

运行一下，试试看，运行结果如下：

```
ret1=项目经理同意小李聚餐费用 300.0 元的请求
ret2=部门经理同意小李聚餐费用 800.0 元的请求
ret3=总经理同意小李聚餐费用 1600.0 元的请求
项目经理同意小张预支差旅费用 3000.0 元的请求
部门经理同意小张预支差旅费用 6000.0 元的请求
总经理同意小张预支差旅费用 36000.0 元的请求
```

仔细体会一下这种设计方式的好处，既通用又灵活。有了新的业务，只需要添加实现新功能的对象就可以了。但是带来的缺陷就是可能会造成对象层次过多，或者出现较多的细粒度的对象。极端情况下，每次扩展一个方法，会出现大量只处理一个功能的细粒度对象。

23. 3. 3　功能链

在实际开发中，经常会遇到把职责链稍稍变形的用法。在标准的职责链中，一个请求在职责链中传递，只要有一个对象处理了这个请求，就会停止。

现在稍稍变通一下，改成一个请求在职责链中传递，每个职责对象负责处理请求的某一方面的功能，处理完成后，不是停止，而是继续向下传递请求，当请求通过很多职责对象处理后，功能也就完成了，把这样的职责链称为功能链。

考虑这样一个功能，在实际应用开发中，进行业务处理之前，通常需要进行权限检查、通用数据校验、数据逻辑校验等处理，然后才开始真正的业务逻辑实现。可以把这些功能分散到一个功能链中。这样做的目的是使程序结构更加灵活，而且复用性会更好。比如通用的权限检查只需要做一份，然后就可以在多个功能链中使用了。

有些朋友看到这里，可能会想，这不是可以使用装饰模式来实现吗？没错，是可以用装饰模式来实现这样的功能，但职责链会更灵活一些。因为装饰模式是在已有的功能上增加新的功能，多个装饰器之间会有一定的联系；而职责链模式的各个职责对象实现的功能，相互之间是没有关联的，是自己实现属于自己处理的那一份功能。

延伸	**可能有些朋友会想到这很类似于在 Web 应用开发中的过滤器 Filter，没错，过滤器链就类似于一个功能链，每个过滤器负责自己的处理，然后转交给下一个过滤器，直到把所有的过滤器都走完，最后进入到 Servlet 中进行处理。最常见的过滤器功能，比如权限检查、字符集转换等，基本上都是 Web 应用的标配。**

接下来在示例中实现这样的功能，实现商品销售的业务处理，在真正进行销售的业务处理之前，需要对传入处理的数据进行权限检查、通用数据检查和数据逻辑检查，只有这些检查都能通过的情况下，才说明传入的数据是正确的、有效的，才可以进行真正的业务功能处理。

（1）首先定义已有的业务功能和封装业务数据的对象，用前面出现过的保存销售信息的业务。为了简单，就不再定义接口了。示例代码如下：

```
/**
/**
 * 商品销售管理模块的业务处理
 */
public class GoodsSaleEbo {
    /**
     * 保存销售信息，本来销售数据应该是多条，太麻烦了，为了演示，简单点
     * @param user 操作人员
     * @param customer 客户
     * @param saleModel 销售数据
     * @return 是否保存成功
     */
    public boolean sale(String user,String customer
                                    ,SaleModel saleModel){
        //如果全部在这里处理，基本的顺序是：
        //1：权限检查
        //2：通用数据检查（这个也可能在表现层已经做过了）
        //3：数据逻辑校验

        //4：真正的业务处理

        //但是现在通过功能链来做，这里就主要负责构建链
        //暂时还没有功能链，等实现好了各个处理对象再回来添加
        return true;
    }
}
```

对应的封装销售数据的对象。示例代码如下：

```
/**
 * 封装销售单的数据，简单地示意一下
 */
public class SaleModel {
    /**
     * 销售的商品
     */
    private String goods;
    /**
     * 销售的数量
     */
    private int saleNum;
    public String getGoods() {
```

```
        return goods;
    }
    public void setGoods(String goods) {
        this.goods = goods;
    }
    public int getSaleNum() {
        return saleNum;
    }
    public void setSaleNum(int saleNum) {
        this.saleNum = saleNum;
    }
    public String toString(){
        return "商品名称="+goods+"，销售数量="+saleNum;
    }
}
```

（2）定义一个用来处理保存销售数据功能的职责对象的接口。示例代码如下：

```
/**
 * 定义职责对象的接口
 */
public abstract class SaleHandler {
    /**
     * 持有下一个处理请求的对象
     */
    protected SaleHandler successor = null;
    /**
     * 设置下一个处理请求的对象
     * @param successor 下一个处理请求的对象
     */
    public void setSuccessor(SaleHandler successor){
        this.successor = successor;
    }
    /**
     * 处理保存销售信息的请求
     * @param user 操作人员
     * @param customer 客户
     * @param saleModel 销售数据
     * @return 是否处理成功
     */
    public abstract boolean sale(String user,String customer,
                                        SaleModel saleModel);
```

```
}
```

（3）实现各个职责处理对象。每个职责对象负责请求的一个方面的处理，把这些职责对象都走完了，功能也就实现完了。

先定义处理安全检查的职责对象。示例代码如下：

```
/**
 * 进行权限检查的职责对象
 */
public class SaleSecurityCheck extends SaleHandler{
    public boolean sale(String user, String customer
                                , SaleModel saleModel) {
        //进行权限检查，简单点，就小李能通过
        if("小李".equals(user)){
            return this.successor.sale(user, customer, saleModel);
        }else{
            System.out.println("对不起"+user
                                +"，你没有保存销售信息的权限");
            return false;
        }
    }
}
```

接下来定义通用数据检查的职责对象。示例代码如下：

```
/**
 * 进行数据通用检查的职责对象
 */
public class SaleDataCheck extends SaleHandler{
    public boolean sale(String user, String customer
                                , SaleModel saleModel) {
        //进行数据通用检查，稍麻烦点，每个数据都要检测
        if(user==null || user.trim().length()==0){
            System.out.println("申请人不能为空");
            return false;
        }
        if(customer==null || customer.trim().length()==0){
            System.out.println("客户不能为空");
            return false;
        }
        if(saleModel==null ){
            System.out.println("销售商品的数据不能为空");
            return false;
        }
```

```
        if(saleModel.getGoods() == null
              || saleModel.getGoods().trim().length()==0){
            System.out.println("销售的商品不能为空");
            return false;
        }
        if(saleModel.getSaleNum()==0){
            System.out.println("销售商品的数量不能为0");
            return false;
        }
        //如果通过了上面的检测，那就向下继续执行
        return this.successor.sale(user, customer, saleModel);
    }
}
```

再看看进行数据逻辑检查的职责对象。示例代码如下：

```
/**
 * 进行数据逻辑检查的职责对象
 */
public class SaleLogicCheck extends SaleHandler{
    public boolean sale(String user, String customer
                                        , SaleModel saleModel) {
        //进行数据的逻辑检查，比如检查 ID 的唯一性，主外键的对应关系等
        //这里应该检查这种主外键的对应关系，比如销售商品是否存在
        //为了演示简单，直接通过吧

        //如果通过了上面的检测，那就向下继续执行
        return this.successor.sale(user, customer, saleModel);
    }
}
```

最后是真正的业务处理的职责对象。示例代码如下：

```
/**
 * 真正处理销售业务功能的职责对象
 */
public class SaleMgr extends SaleHandler{
    public boolean sale(String user, String customer
                                      , SaleModel saleModel) {
        //进行真正的业务逻辑处理
        System.out.println(user+"保存了"+customer
                            +"购买 "+saleModel+" 的销售数据");
        return true;
    }
```

```
}
```

（4）实现了各个职责对象处理，回过头来看看如何具体实现业务处理，在业务对象中进行功能链的组合。示例代码如下：

```
public class GoodsSaleEbo {
    /**
     * 保存销售信息，本来销售数据应该是多条，太麻烦了，为了演示，简单点
     * @param user 操作人员
     * @param customer 客户
     * @param saleModel 销售数据
     * @return 是否保存成功
     */
    public boolean sale(String user,String customer
                                    ,SaleModel saleModel){
        //如果全部在这里处理，基本的顺序是
        //1：权限检查
        //2：通用数据检查（这个也可能在表现层已经做过了）
        //3：数据逻辑校验

        //4：真正的业务处理

        //但是现在通过功能链来做，这里主要负责构建链
        SaleSecurityCheck ssc = new SaleSecurityCheck();
        SaleDataCheck sdc = new SaleDataCheck();
        SaleLogicCheck slc = new SaleLogicCheck();
        SaleMgr sd = new SaleMgr();
        ssc.setSuccessor(sdc);
        sdc.setSuccessor(slc);
        slc.setSuccessor(sd);
        //向链上的第一个对象发出处理的请求
        return ssc.sale(user, customer, saleModel);
    }
}
```

（5）写个客户端，调用业务对象，测试一下看看。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建业务对象
        GoodsSaleEbo ebo = new GoodsSaleEbo();
        //准备测试数据
        SaleModel saleModel = new SaleModel();
        saleModel.setGoods("张学友怀旧经典");
```

```
        saleModel.setSaleNum(10);

        //调用业务功能
        ebo.sale("小李", "张三", saleModel);
        ebo.sale("小张", "李四", saleModel);
    }
}
```

运行一下，试试看。运行结果如下：

```
小李保存了张三购买 商品名称=张学友怀旧经典，销售数量=10 的销售数据
对不起小张，你没有保存销售信息的权限
```

你还可以改变测试的数据，看看效果，好好体会这种设计的灵活性。很多框架级功能的设计都用得上。

23.3.4　职责链模式的优缺点

职责链模式有以下优点。

- 请求者和接收者松散耦合

  在职责链模式中，请求者并不知道接收者是谁，也不知道具体如何处理，请求者只是负责向职责链发出请求就可以了。而每个职责对象也不用管请求者或者是其他的职责对象，只负责处理自己的部分，其他的就交给其他的职责对象去处理。也就是说，请求者和接收者是完全解耦的。

- 动态组合职责

  职责链模式会把功能处理分散到单独的职责对象中，然后在使用的时候，可以动态组合职责形成职责链，从而可以灵活地给对象分配职责，也可以灵活地实现和改变对象的职责。

职责链模式有以下缺点。

- 产生很多细粒度对象

  职责链模式会把功能处理分散到单独的职责对象中，也就是每个职责对象只处理一个方面的功能，要把整个业务处理完，需要很多职责对象的组合，这样会产生大量的细粒度职责对象。

- 不一定能被处理

  职责链模式的每个职责对象只负责自己处理的那一部分，因此可能会出现某个请求，把整个链传递完了，都没有职责对象处理它。这就需要在使用职责链模式的时候，需要提供默认的处理，并且注意构建的链的有效性。

23.3.5　思考职责链模式

1. 职责链模式的本质

> 职责链模式的本质：**分离职责，动态组合。**

分离职责是前提，只有先把复杂的功能分开，拆分成很多的步骤和小的功能处理，然后才能合理规划和定义职责类。可以有很多的职责类来负责处理某一个功能，让每个职责类负责处理功能的某一个方面，在运行期间进行动态组合，形成一个处理的链，把这个链运行完，功能也就处理完了。

动态组合才是职责链模式的精华所在，因为要实现请求对象和处理对象的解耦，请求对象不知道谁才是真正的处理对象，因此要动态地把可能的处理对象组合起来。由于组合的方式是动态的，这就意味着可以很方便地修改和添加新的处理对象，从而让系统更加灵活和具有更好的扩展性。

> 延伸 **当然这么做还会有一个潜在的优点，就是可以增强职责功能的复用性。如果职责功能是很多地方都可以使用的公共功能，那么它可以在多个职责链中复用。**

2. 何时选用职责链模式

建议在以下情况中选用职责链模式。

- 如果有多个对象可以处理同一个请求，但是具体由哪个对象来处理该请求，是运行时刻动态确定的。这种情况可以使用职责链模式，把处理请求的对象实现成为职责对象，然后把它们构成一个职责链，当请求在这个链中传递的时候，具体由哪个职责对象来处理，会在运行时动态判断。
- 如果你想在不明确指定接收者的情况下，向多个对象中的其中一个提交请求的话，可以使用职责链模式。职责链模式实现了请求者和接收者之间的解耦，请求者不需要知道究竟是哪一个接收者对象来处理了请求。
- 如果想要动态指定处理一个请求的对象集合，可以使用职责链模式。职责链模式能动态地构建职责链，也就是动态地来决定到底哪些职责对象来参与到处理请求中来，相当于是动态地指定了处理一个请求的职责对象集合。

23.3.6 相关模式

- 职责链模式和组合模式

  这两个模式可以组合使用。

  可以把职责对象通过组合模式来组合，这样可以通过组合对象自动递归地向上调用，由父组件作为子组件的后继，从而形成链。

  这也就是前面提到过的使用外部已有的链接，这种情况在客户端使用的时候，就不用再构造链了，虽然不构造链，但是需要构造组合对象树，是一样的。
- 职责链模式和装饰模式

  这两个模式相似，从某个角度讲，可以相互模拟实现对方的功能。

  装饰模式能够动态地给被装饰对象添加功能，要求装饰器对象和被装饰的对象实现相同的接口。而职责链模式可以实现动态的职责组合，标准的功能是有一个对象处理就结束，但是如果处理完本职责不急于结束，而是继续向下传递请求，那么其功能就和装饰模式的功能差不多了，每个职责对象就类似于装饰器，

可以实现某种功能。

而且两个模式的本质也相似，都需要在运行期间动态组合，装饰模式是动态组合装饰器，而职责链是动态组合处理请求的职责对象的链。

但是从标准的设计模式上来讲，这两个模式还是有较大区别的，这点要注意。首先是目的不同，装饰模式是要实现透明的为对象添加功能，而职责链模式是要实现发送者和接收者解耦；另外一个，装饰模式是无限递归调用的，可以有任意多个对象来装饰功能，但是职责链模式是有一个处理就结束。

- 职责链模式和策略模式

这两个模式可以组合使用。

这两个模式有相似之处，如果把职责链简化到直接就能选择到相应的处理对象，那就跟策略模式的选择差不多，因此可以用职责链来模拟策略模式的功能。只是如果把职责链简化到这个地步，也就不存在链了，也就称不上是职责链了。

两个模式可以组合使用，可以在职责链模式的某个职责实现的时候，使用策略模式来选择具体的实现，同样也可以在策略模式的某个策略实现中，使用职责链模式来实现功能处理。

同理职责链模式也可以和状态模式组合使用。

读书笔记

第 24 章　桥接模式（Bridge）

24.1 场景问题

24.1.1 发送提示消息

考虑这样一个实际的业务功能：发送提示消息。基本上所有带业务流程处理的系统都会有这样的功能，比如某人有新的工作了，需要发送一条消息提示他。

从业务上看，消息又分成普通消息、加急消息和特急消息多种，不同的消息类型，业务功能处理是不一样的，比如加急消息是在消息上添加加急，而特急消息除了添加特急外，还会做一条催促的记录，多久不完成会继续催促；从发送消息的手段上看，又有系统内短消息、手机短消息、邮件等。

现在要实现这样的发送提示消息的功能，该如何实现呢？

24.1.2 不用模式的解决方案

1. 实现简化版本

先考虑实现一个简单点的版本，比如，消息只是实现发送普通消息，发送的方式呢，只实现系统内短消息和邮件。其他的功能，等这个版本完成后，再继续添加。这样先把问题简单化，实现起来会容易一点。

由于发送普通消息会有两种不同的实现方式，为了让外部能统一操作，因此，把消息设计成接口，然后由两个不同的实现类分别实现系统内短消息方式和邮件发送消息的方式。此时系统结构如图 24.1 所示。

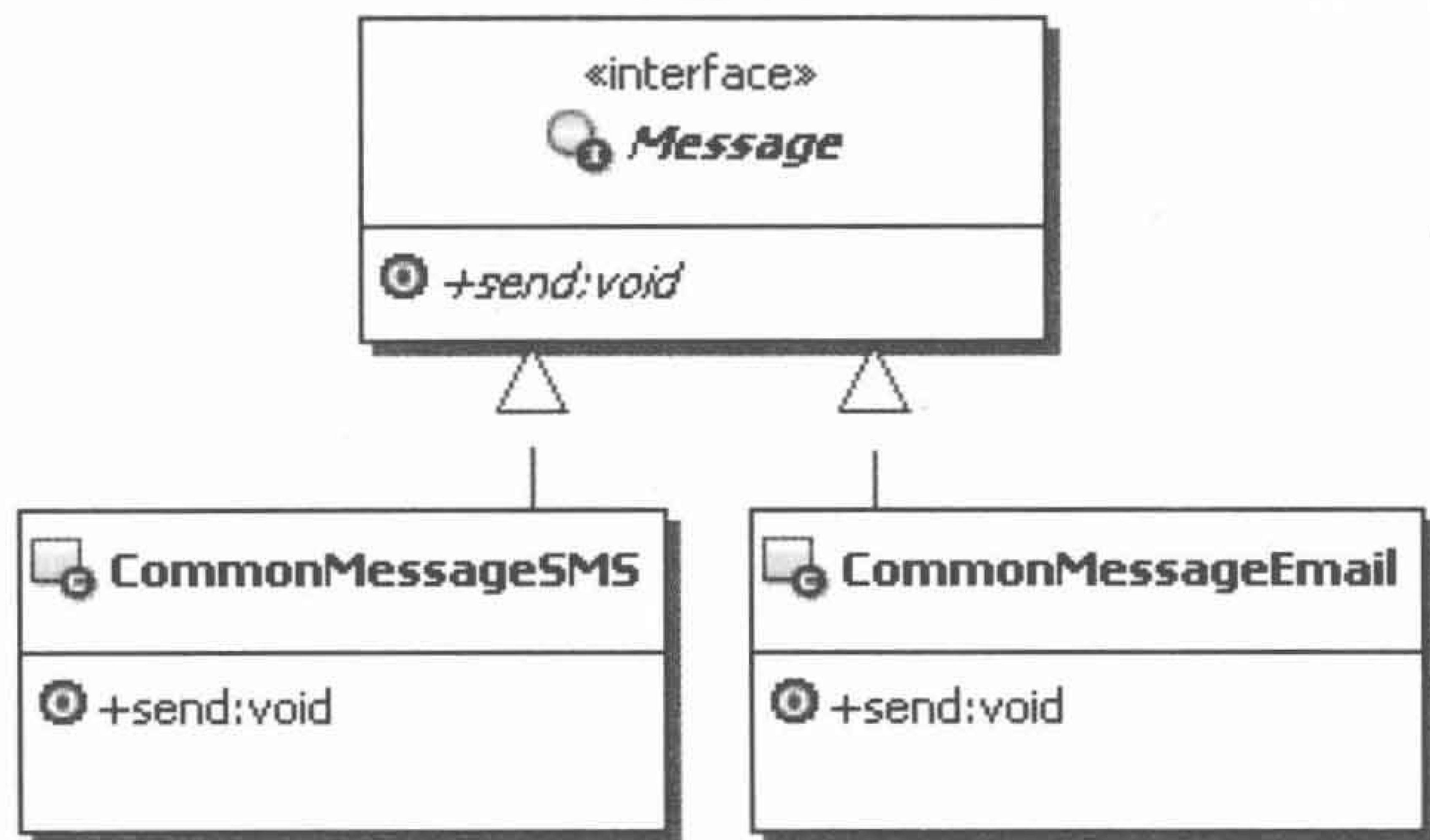


图 24.1　简化版本的系统结构示意图

下面看看大致的实现示意。

（1）先来看看消息的统一接口。示例代码如下：

```
/**
 * 消息的统一接口
 */
public interface Message {
```

```
    /**
     * 发送消息
     * @param message 要发送的消息内容
     * @param toUser 消息发送的目的人员
     */
    public void send(String message,String toUser);
}
```

（2）再来分别看看两种实现方式。这里只是为了示意，并不会真的去发送 E-mail 和站内短消息。

先看看站内短消息的方式。示例代码如下：

```
/**
 * 以站内短消息的方式发送普通消息
 */
public  class CommonMessageSMS implements Message{
    public void send(String message, String toUser) {
        System.out.println("使用站内短消息的方式，发送消息'"
                                    +message+"'给"+toUser);
    }
}
```

同样地，实现以 E-mail 的方式发送普通消息。示例代码如下：

```
/**
 * 以E-mail的方式发送普通消息
 */
public class CommonMessageEmail implements Message{
    public void send(String message, String toUser) {
        System.out.println("使用E-mail的方式，发送消息'"
                                    +message+"'给"+toUser);
    }
}
```

2. 实现发送加急消息

上面的实现看起来很简单，对不对？接下来，添加发送加急消息的功能，也有两种发送的方式，同样是站内短消息和 E-mail 的方式。

加急消息的实现不同于普通消息。加急消息会自动在消息上添加加急，然后再发送消息；另外加急消息会提供监控的方法，让客户端可以随时通过这个方法来了解对于加急消息处理的进度，比如，相应的人员是否接收到这个信息，相应的工作是否已经开展等。因此加急消息需要扩展出一个新的接口，除了基本的发送消息的功能，还需要添加监控的功能。这个时候，系统的结构如图 24.2 所示。

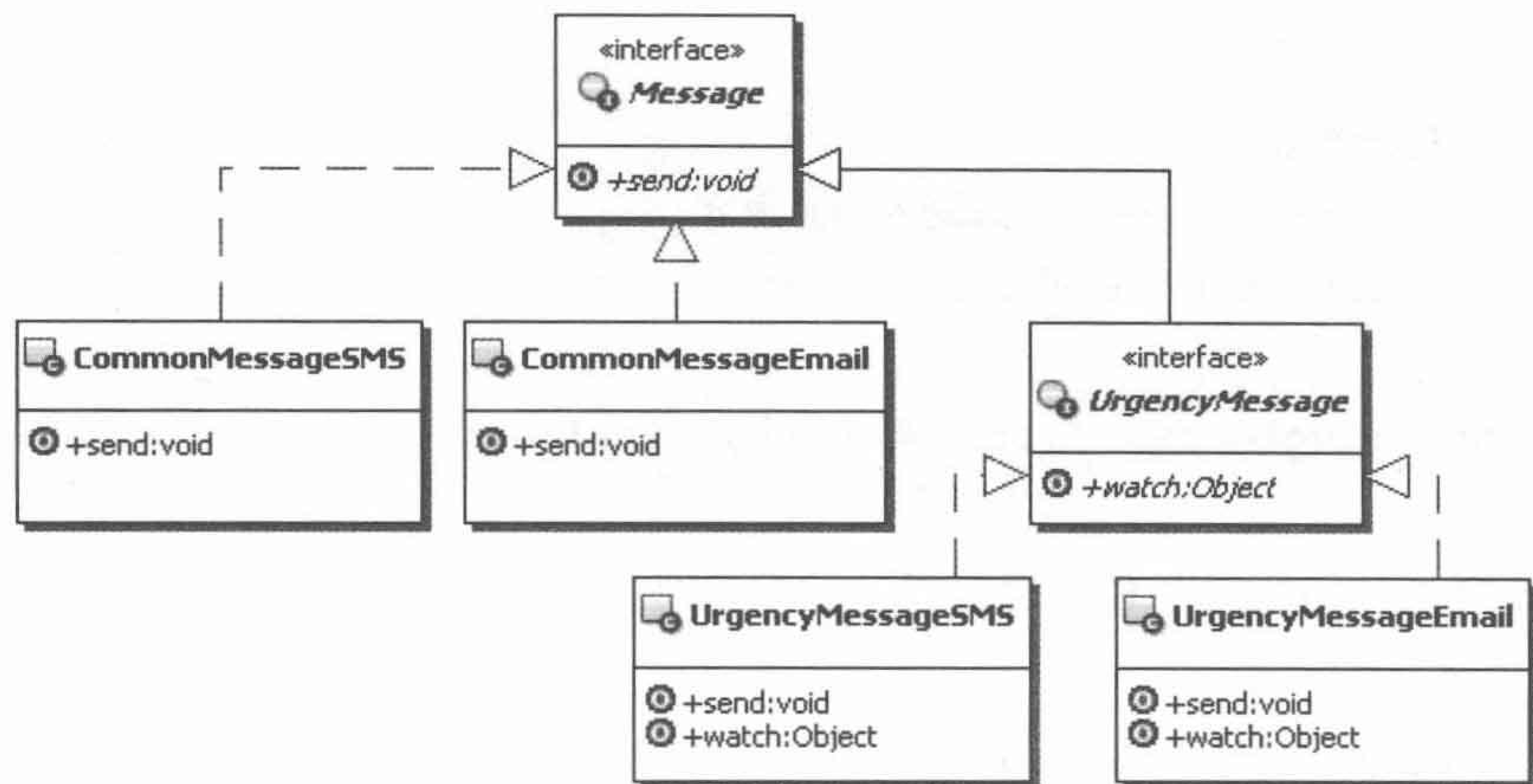


图 24.2　加入发送加急消息后的系统结构示意图

（1）先看看扩展出来的加急消息的接口。示例代码如下：

```
/**
 * 加急消息的抽象接口
 */
public interface UrgencyMessage extends Message{
    /**
     * 监控某消息的处理过程
     * @param messageId 被监控的消息的编号
     * @return 包含监控到的数据对象，这里示意一下，所以用了 Object
     */
    public Object watch(String messageId);
}
```

（2）相应的实现方式还是发送站内短消息和 E-mail 两种，同样需要两个实现类来分别实现这两种方式。

先看看站内短消息的方式。示例代码如下：

```
public class UrgencyMessageSMS implements UrgencyMessage{
    public void send(String message, String toUser) {
        message = "加急："+message;
        System.out.println("使用站内短消息的方式，发送消息'"
                                            +message+"'给"+toUser);
    }

    public Object watch(String messageId) {
        //获取相应的数据，组织成监控的数据对象，然后返回
        return null;
    }
}
```

再看看 E-mail 的方式。示例代码如下：

```
public class UrgencyMessageEmail implements UrgencyMessage{
    public void send(String message, String toUser) {
        message = "加急："+message;
        System.out.println("使用E-mail的方式，发送消息'"
                                        +message+"'给"+toUser);
    }
    public Object watch(String messageId) {
        //获取相应的数据，组织成监控的数据对象，然后返回
        return null;
    }
}
```

事实上，在实现加急消息发送的功能上，可能会使用前面发送不同消息的功能，也就是让实现加急消息处理的对象继承普通消息的相应实现。但这里为了让结构简单、清晰一点，所以没有这样做。

24.1.3　有何问题

上面这样实现，好像也能满足基本的功能要求，可是这么实现好不好呢？有没有什么问题呢？

咱们继续向下来添加功能实现。为了简洁，就不再进行代码示意了，通过实现的结构示意图就可以看出实现上的问题。

1. 继续添加特急消息的处理

特急消息不需要查看处理进程，只要没有完成，就直接催促，也就是说，对于特急消息，在普通消息的处理基础上，需要添加催促的功能。而特急消息和催促的发送方式，相应的实现方式还是发送站内短消息和 E-mail 两种。此时系统的结构如图 24.3 所示。

仔细观察上面的系统结构示意图，会发现一个很明显的问题，那就是通过这种继承的方式来扩展消息处理，会非常不方便。

会看到在实现加急消息处理的时候，必须实现站内短消息和 E-mail 两种处理方式，因为业务处理可能不同；在实现特急消息处理的时候，又必须实现站内短消息和 E-mail 这两种处理方式。

这意味着，以后每次扩展一下消息处理，都必须要实现这两种处理方式，是不是很痛苦？这还不算完，如果要添加新的实现方式呢？继续向下看吧。

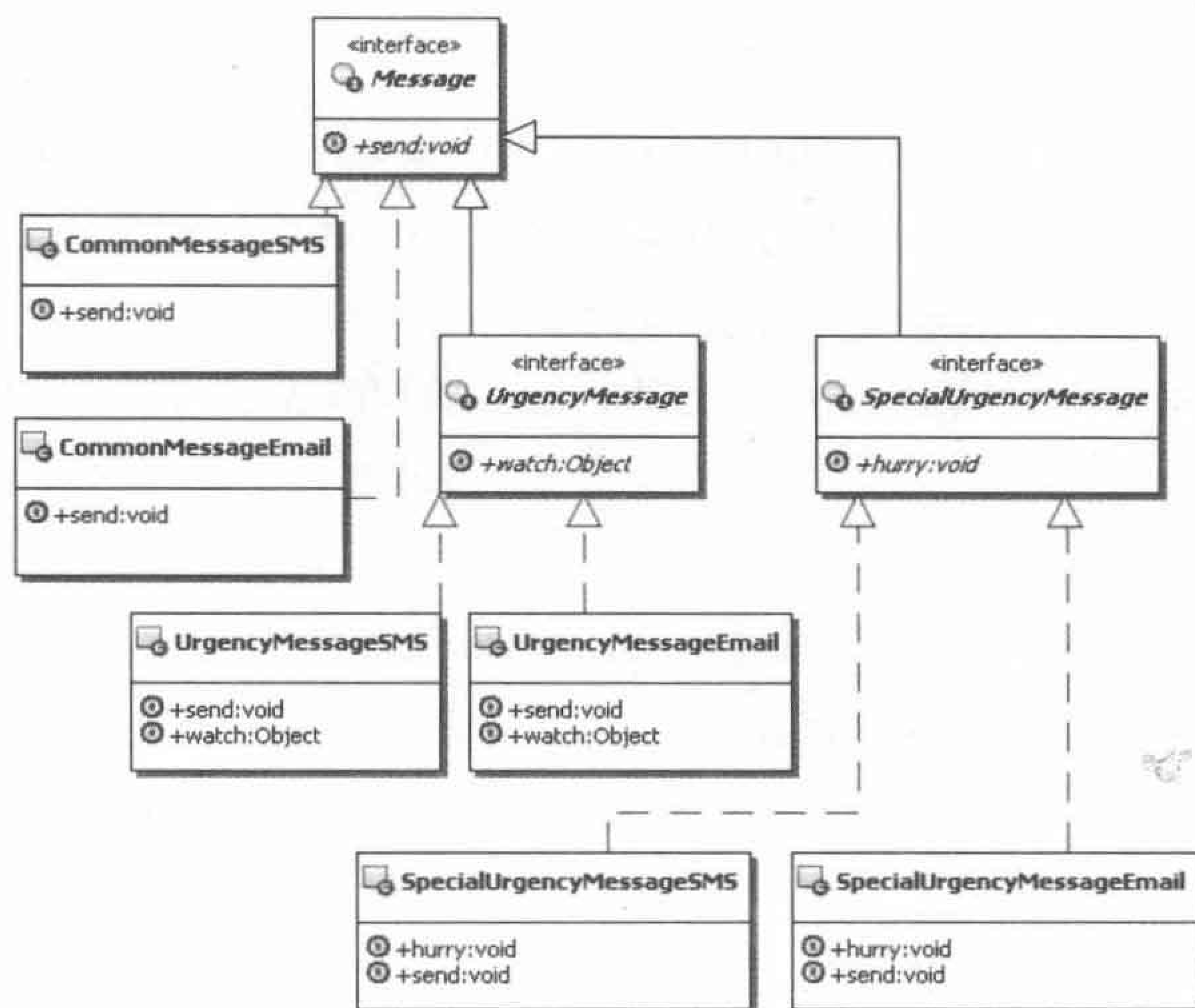


图 24.3　加入发送特急消息后的系统结构示意图

2. 继续添加发送手机消息的处理方式

如果看到上面的实现，你还感觉问题不是很大的话，继续完成功能，添加发送手机消息的处理方式。

仔细观察现在的实现，如果要添加一种新的发送消息的方式，是需要在每一种抽象的具体实现中，都要添加发送手机消息的处理的。也就是说，发送普通消息、加急消息和特急消息的处理，都可以通过手机来发送。这就意味着，需要添加三个实现。此时系统结构如图 24.4 所示。

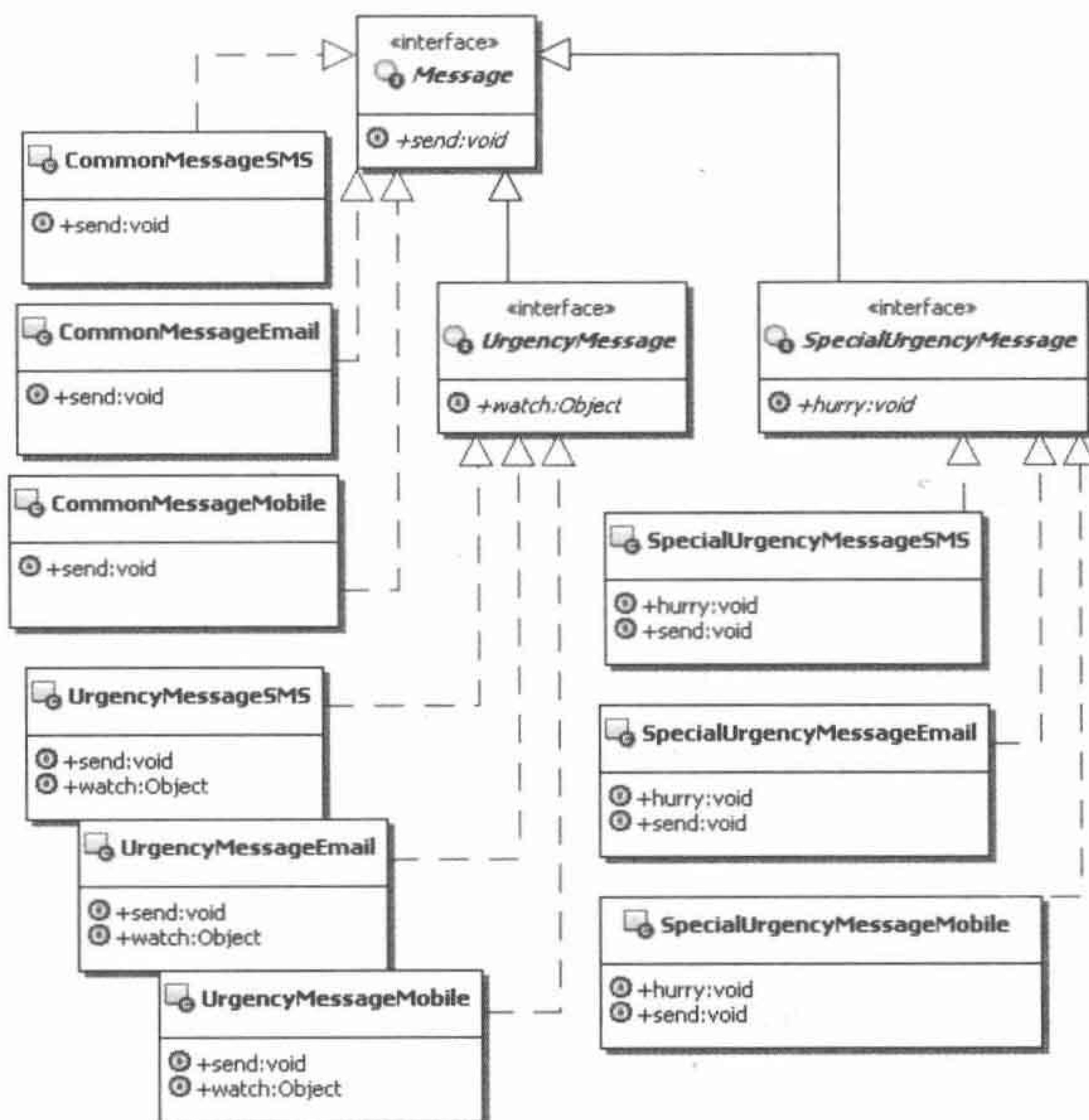


图 24.4　加入发送手机消息后的系统结构示意图

现在体会到这种实现方式的大问题了吧。

3. 小结一下出现的问题

采用通过继承来扩展的实现方式，有个明显的缺点，扩展消息的种类不太容易。不

同种类的消息具有不同的业务，也就是有不同的实现，在这种情况下，每个种类的消息，需要实现所有不同的消息发送方式。

更可怕的是，如果要新加入一种消息的发送方式，那么会要求所有的消息种类都要加入这种新的发送方式的实现。

要是考虑业务功能上再扩展一下呢？比如，要求实现群发消息，也就是一次可以发送多条消息，这就意味着很多地方都得修改，太恐怖了。

那么究竟该如何实现才能既实现功能，又能灵活地扩展呢？

24.2 解决方案

24.2.1 使用桥接模式来解决问题

用来解决上述问题的一个合理的解决方案，就是使用桥接模式。那么什么是桥接模式呢？

1. 桥接模式的定义

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

2. 应用桥接模式来解决问题的思路

仔细分析上面的示例，根据示例的功能要求，示例的变化具有两个纬度，一个纬度是抽象的消息这边，包括普通消息、加急消息和特急消息，这几个抽象的消息本身就具有一定的关系，加急消息和特急消息会扩展普通消息；另一个纬度是在具体的消息发送方式上，包括站内短消息、E-mail 和手机短信息，这几个方式是平等的，可被切换的方式。这两个纬度一共可以组合出 9 种不同的可能性来。它们的关系如图 24.5 所示。

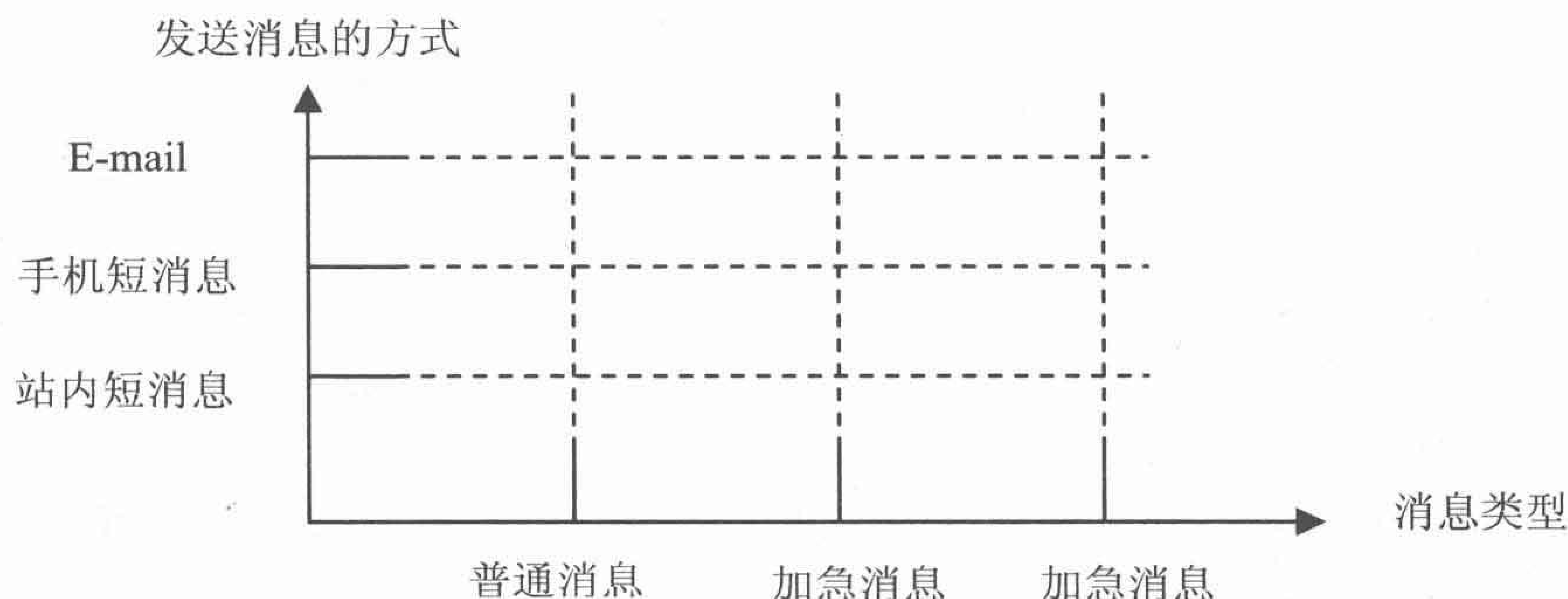


图 24.5 发送消息的可能性的组合示意图

> 提示　**现在出现问题的根本原因，就在于消息的抽象和实现是混杂在一起的，这就导致了一个纬度的变化会引起另一个纬度进行相应的变化，从而使得程序扩展起来非常困难。**

要想解决这个问题，就必须把这两个纬度分开，也就是将抽象部分和实现部分分开，让它们相互独立，这样就可以实现独立的变化，使扩展变得简单。

桥接模式通过引入实现的接口，把实现部分从系统中分离出去。那么，抽象这边如何使用具体的实现呢？肯定是用面向实现的接口来编程了，为了让抽象这边能够很方便地与实现结合起来，把顶层的抽象接口改成抽象类，在其中持有一个具体的实现部分的实例。

这样一来，对于需要发送消息的客户端而言，就只需要创建相应的消息对象，然后调用这个消息对象的方法就可以了，这个消息对象会调用持有的真正的消息发送方式来把消息发送出去。也就是说客户端只是想要发送消息而已，并不想关心具体如何发送。

24.2.2　桥接模式的结构和说明

桥接模式的结构如图 24.6 所示。

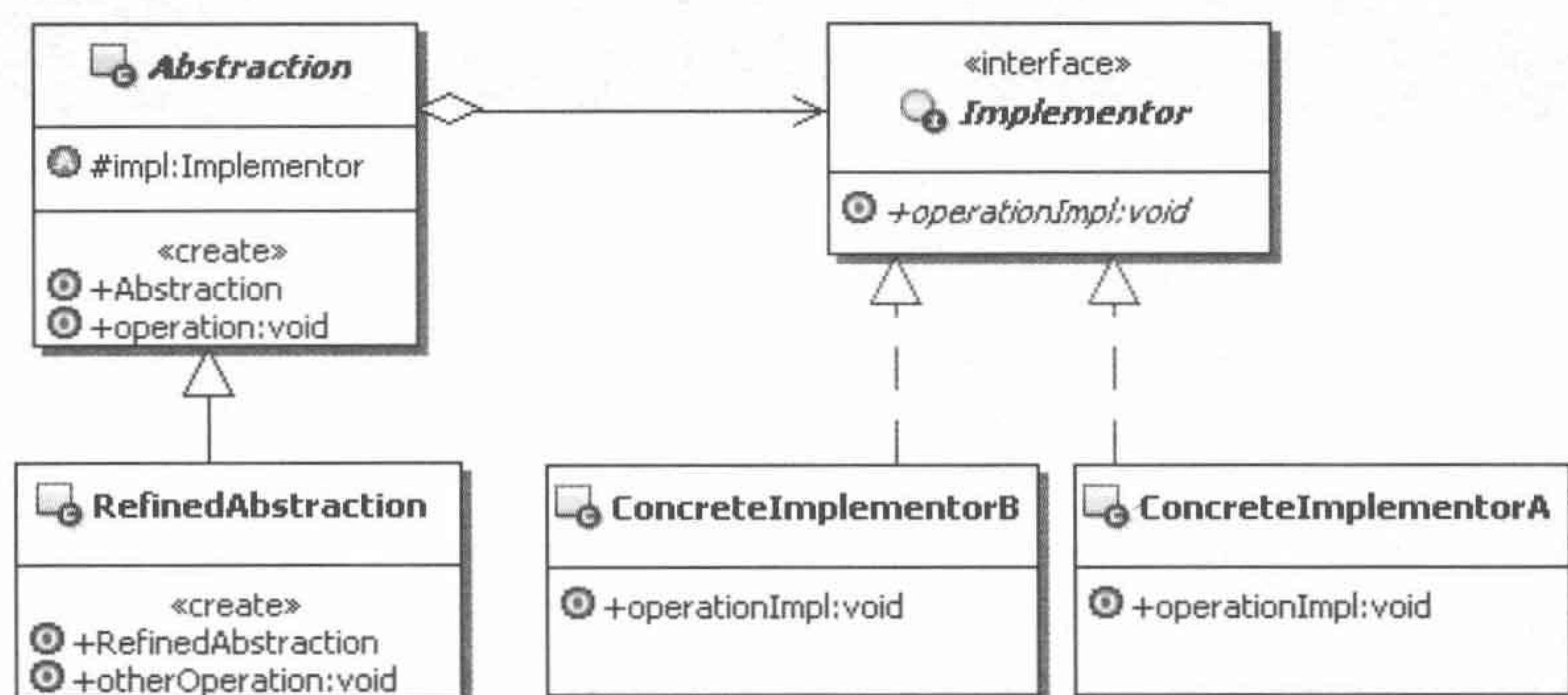


图 24.6　桥接模式的结构示意图

- Abstraction：抽象部分的接口。通常在这个对象中，要维护一个实现部分的对象引用，抽象对象里面的方法，需要调用实现部分的对象来完成。这个对象中的方法，通常都是和具体的业务相关的方法。
- RefinedAbstraction：扩展抽象部分的接口。通常在这些对象中，定义跟实际业务相关的方法，这些方法的实现通常会使用 Abstraction 中定义的方法，也可能需要调用实现部分的对象来完成。
- Implementor：定义实现部分的接口。这个接口不用和 Abstraction 中的方法一致，通常是由 Implementor 接口提供基本的操作。而 Abstraction 中定义的是基于这些基本操作的业务方法，也就是说 Abstraction 定义了基于这些基本操作的较高层次的操作。
- ConcreteImplementor：真正实现 Implementor 接口的对象。

24.2.3　桥接模式示例代码

（1）先看看 Implementor 接口的定义。示例代码如下：

```
/**
 * 定义实现部分的接口，可以与抽象部分接口的方法不一样
 */
public interface Implementor {
    /**
     * 示例方法，实现抽象部分需要的某些具体功能
     */
    public void operationImpl();
}
```

（2）再看看 Abstraction 接口的定义。注意一点，虽然说是接口定义，但其实是实现成为抽象类。示例代码如下：

```
/**
 * 定义抽象部分的接口
 */
public abstract class Abstraction {
    /**
     * 持有一个实现部分的对象
     */
    protected Implementor impl;
    /**
     * 构造方法，传入实现部分的对象
     * @param impl 实现部分的对象
     */
    public Abstraction(Implementor impl){
        this.impl = impl;
    }
    /**
     * 示例操作，实现一定的功能，可能需要转调实现部分的具体实现方法
     */
    public void operation() {
        impl.operationImpl();
    }
}
```

（3）接下来看看具体的实现。其中一个实现的示例代码如下：

```
/**
 * 真正的具体实现对象
```

```
 */
public class ConcreteImplementorA implements Implementor {
    public void operationImpl() {
        //真正的实现
    }
}
```

另外一个实现。示例代码如下：

```
/**
 * 真正的具体实现对象
 */
public class ConcreteImplementorB implements Implementor {
    public void operationImpl() {
        //真正的实现
    }
}
```

（4）最后来看看扩展 Abstraction 接口的对象实现。示例代码如下：

```
/**
 * 扩充由 Abstraction 定义的接口功能
 */
public class RefinedAbstraction extends Abstraction {
    public RefinedAbstraction(Implementor impl) {
        super(impl);
    }
    /**
     * 示例操作，实现一定的功能
     */
    public void otherOperation(){
        //实现一定的功能，可能会使用具体实现部分的实现方法
        //但是本方法更大的可能是使用 Abstraction 中定义的方法
        //通过组合使用 Abstraction 中定义的方法来完成更多的功能
    }
}
```

24.2.4 使用桥接模式重写示例

学习了桥接模式的基础知识，该来使用桥接模式重写前面的示例了。通过示例，来看看使用桥接模式来实现同样的功能，是否能解决“既能方便地实现功能，又能有很好的扩展性”的问题。

要使用桥接模式来重新实现前面的示例，首要任务就是要把抽象部分和实现部分分

离出来，分析要实现的功能。抽象部分就是各个消息的类型所对应的功能，而实现部分就是各种发送消息的方式。

其次要按照桥接模式的结构，给抽象部分和实现部分分别定义接口，然后分别实现它们就可以了。

1. 从简单功能开始

从相对简单的功能开始，先实现普通消息和加急消息的功能，发送方式先实现站内短消息和 E-mail 两种。

使用桥接模式来实现这些功能的程序结构如图 24.7 所示。

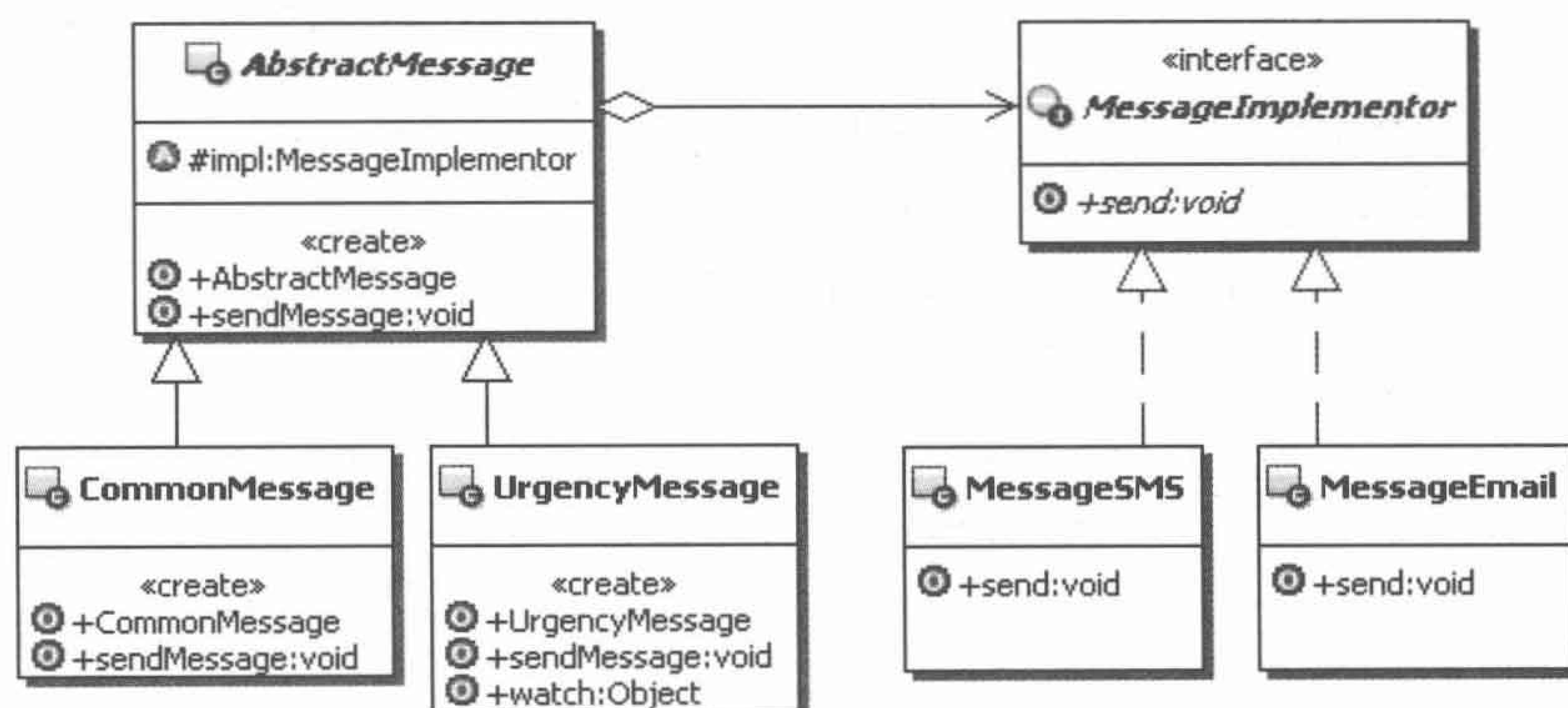


图 24.7　使用桥接模式来实现简单功能示例的程序结构示意图

下面看看代码实现，会更清楚一些。

（1）先看看实现部分定义的接口。示例代码如下：

```
/**
 * 实现发送消息的统一接口
 */
public interface MessageImplementor {
    /**
     * 发送消息
     * @param message 要发送的消息内容
     * @param toUser 消息发送的目的人员
     */
    public void send(String message,String toUser);
}
```

（2）再看看抽象部分定义的接口。示例代码如下：

```
/**
 * 抽象的消息对象
 */
public abstract class AbstractMessage {
    /**
     * 持有一个实现部分的对象
```

```
    */
    protected MessageImplementor impl;
    /**
     * 构造方法，传入实现部分的对象
     * @param impl 实现部分的对象
     */
    public AbstractMessage(MessageImplementor impl){
        this.impl = impl;
    }
    /**
     * 发送消息，转调实现部分的方法
     * @param message 要发送的消息内容
     * @param toUser 消息发送的目的人员
     */
    public void sendMessage(String message,String toUser){
        this.impl.send(message, toUser);
    }
}
```

（3）看看如何具体地实现发送消息。

先看看站内短消息的实现吧。示例代码如下：

```
/**
 * 以站内短消息的方式发送消息
 */
public  class MessageSMS implements MessageImplementor{
    public void send(String message, String toUser) {
        System.out.println("使用站内短消息的方式，发送消息'"
                                              +message+"'给"+toUser);
    }
}
```

再看看 E-mail 方式的实现。示例代码如下：

```
/**
 * 以 E-mail 的方式发送消息
 */
public class MessageEmail implements MessageImplementor{
    public void send(String message, String toUser) {
        System.out.println("使用E-mail的方式，发送消息'"
                                              +message+"'给"+toUser);
    }
}
```

（4）接下来该看看如何扩展抽象的消息接口了。

先看看普通消息的实现。示例代码如下：

```
public class CommonMessage extends AbstractMessage{
    public CommonMessage(MessageImplementor impl) {
        super(impl);
    }
    public void sendMessage(String message, String toUser) {
        //对于普通消息，什么都不干，直接调用父类的方法，把消息发送出去就可以了
        super.sendMessage(message, toUser);
    }
}
```

再看看加急消息的实现。示例代码如下：

```
public class UrgencyMessage extends AbstractMessage{
    public UrgencyMessage(MessageImplementor impl) {
        super(impl);
    }
    public void sendMessage(String message, String toUser) {
        message = "加急："+message;
        super.sendMessage(message, toUser);
    }
    /**
     * 扩展自己的新功能：监控某消息的处理过程
     * @param messageId 被监控的消息的编号
     * @return 包含监控到的数据对象，这里示意一下，所以用了 Object
     */
    public Object watch(String messageId) {
        //获取相应的数据，组织成监控的数据对象，然后返回
        return null;
    }
}
```

2. 添加功能

看了上面的实现，发现使用桥接模式来实现也不是很困难，关键得看是否能解决前面提出的问题，那就来添加还未实现的功能看看，添加对特急消息的处理，同时添加一个使用手机发送消息的方式，该怎么实现呢？

很简单，只需要在抽象部分再添加一个特急消息的类，扩展抽象消息就可以把特急消息的处理功能加入到系统中；对于添加手机发送消息的方式也很简单，在实现部分新增加一个实现类，实现用手机发送消息的方式就可以了。

这么简单？好像看起来完全没有了前面所提到的问题。 的确如此，采用桥接模式来

实现，抽象部分和实现部分分离开了，可以相互独立地变化，而不会相互影响。因此在抽象部分添加新的消息处理，对发送消息的实现部分是没有影响的；反过来增加发送消息的方式，对消息处理部分也是没有影响的。

接着看看代码实现。

（1）先看看新的特急消息的处理类。示例代码如下：

```
public class SpecialUrgencyMessage extends AbstractMessage{
    public SpecialUrgencyMessage(MessageImplementor impl) {
        super(impl);
    }
    public void hurry(String messageId) {
        //执行催促的业务，发出催促的信息
    }
    public void sendMessage(String message, String toUser) {
        message = "特急："+message;
        super.sendMessage(message, toUser);
        //还需要增加一条待催促的信息
    }
}
```

（2）再看看使用手机短消息的方式发送消息的实现。示例代码如下：

```
/**
 * 以手机短消息的方式发送消息
 */
public  class MessageMobile implements MessageImplementor{
    public void send(String message, String toUser) {
        System.out.println("使用手机短消息的方式，发送消息'"
                                    +message+"'给"+toUser);
    }
}
```

3. 测试功能

看了上面的实现，可能会感觉到，使用桥接模式来实现前面的示例，添加新的消息处理，或者是新的消息发送方式是如此简单，可是这样实现，好用吗？写个客户端来测试和体会一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建具体的实现对象
        MessageImplementor impl = new MessageSMS();
        //创建一个普通消息对象
        AbstractMessage m = new CommonMessage(impl);
```

```
            m.sendMessage("请喝一杯茶", "小李");
            //创建一个紧急消息对象
            m = new UrgencyMessage(impl);
            m.sendMessage("请喝一杯茶", "小李");
            //创建一个特急消息对象
            m = new SpecialUrgencyMessage(impl);
            m.sendMessage("请喝一杯茶", "小李");

            //把实现方式切换成手机短消息，然后再实现一遍
            impl = new MessageMobile();
            m = new CommonMessage(impl);
            m.sendMessage("请喝一杯茶", "小李");
            m = new UrgencyMessage(impl);
            m.sendMessage("请喝一杯茶", "小李");
            m = new SpecialUrgencyMessage(impl);
            m.sendMessage("请喝一杯茶", "小李");
        }
    }
```

运行结果如下：

```
使用站内短消息的方式，发送消息'请喝一杯茶'给小李
使用站内短消息的方式，发送消息'加急：请喝一杯茶'给小李
使用站内短消息的方式，发送消息'特急：请喝一杯茶'给小李
使用手机短消息的方式，发送消息'请喝一杯茶'给小李
使用手机短消息的方式，发送消息'加急：请喝一杯茶'给小李
使用手机短消息的方式，发送消息'特急：请喝一杯茶'给小李
```

前面三条是使用的站内短消息，后面三条是使用的手机短消息，正确地实现了预期的功能。看来前面的实现应该是正确的，能够完成功能，并且能灵活扩展。

24.3 模式讲解

24.3.1 认识桥接模式

1. 什么是桥接

在桥接模式中，不太好理解的就是桥接的概念。什么是桥接？为何需要桥接？如何桥接？把这些问题搞清楚了，也就基本明白桥接的含义了。

一个一个来，先看看什么是桥接？所谓桥接，通俗点说就是在不同的东西之间搭一个桥，让它们能够连接起来，可以相互通讯和使用。那么在桥接模式中到底是给什么东西来搭桥呢？就是为被分离了的抽象部分和实现部分来搭桥，比如前面示例中在抽象的

消息和具体消息发送之间搭个桥。

注意	**但是这里要注意一个问题，在桥接模式中的桥接是单向的，也就是只能是抽象部分的对象去使用具体实现部分的对象，而不能反过来，也就是个单向桥。**

2. 为何需要桥接

为了达到让抽象部分和实现部分都可以独立变化的目的，在桥接模式中，是把抽象部分和实现部分分离开来的，虽然从程序结构上是分开了，但是在抽象部分实现的时候，还是需要使用具体的实现的，这可怎么办呢？抽象部分如何才能调用到具体实现部分的功能呢？很简单，搭个桥就可以了。搭个桥，让抽象部分通过这个桥就可以调用到实现部分的功能了，因此需要桥接。

3. 如何桥接

这个在理解上也很简单，只要让抽象部分拥有实现部分的接口对象，就桥接上了，在抽象部分即可通过这个接口来调用具体实现部分的功能。也就是说，桥接在程序上体现了在抽象部分拥有实现部分的接口对象，维护桥接就是维护这个关系。

4. 独立变化

桥接模式的意图是使得抽象和实现可以独立变化，都可以分别扩充。也就是说抽象部分和实现部分是一种非常松散的关系。从某个角度来讲，抽象部分和实现部分是可以完全分开的，独立的，抽象部分不过是一个使用实现部分对外接口的程序罢了。

延伸	**如果这么看桥接模式的话，就类似于策略模式了。抽象部分需要根据某个策略，来选择真实的实现，也就是说桥接模式的抽象部分相当于策略模式的上下文。更原始的就直接类似于面向接口编程，通过接口分离的两个部分而已。但是别忘了，桥接模式的抽象部分，是可以继续扩展和变化的，而策略模式只有上下文，是不存在所谓抽象部分的。**

那抽象和实现为何还要组合在一起呢？原因是在抽象部分和实现部分还是存在内部联系的，抽象部分的实现通常是需要调用实现部分的功能来实现的。

5. 动态变换功能

由于桥接模式中的抽象部分和实现部分是完全分离的，因此可以在运行时动态组合具体的真实实现，从而达到动态变换功能的目的。

从另外一个角度看，抽象部分和实现部分没有固定的绑定关系，因此同一个真实实现可以被不同的抽象对象使用；反过来，同一个抽象也可以有多个不同的实现。就像前面示例的那样，比如，站内短消息的实现功能，可以被普通消息、加急消息或是特急消息等不同的消息对象使用；反过来，某个消息具体的发送方式，可以是站内短消息或者是 E-mail，也可以是手机短消息等具体的发送方式。

6. 退化的桥接模式

如果 Implementor 仅有一个实现，那么就没有必要创建 Implementor 接口了，这是一种桥接模式退化的情况。这个时候 Abstraction 和 Implementor 是一对一的关系，虽然如

此，也还是要保持它们的分离状态，这样的话，它们才不会相互影响，才可以分别扩展。

也就是说，就算不要 Implementor 接口了，也要保持 Abstraction 和 Implementor 是分离的，模式的分离机制仍然是非常有用的。

7. 桥接模式和继承

继承是扩展对象功能的一种常见手段，通常情况下，继承扩展的功能变化纬度都是一纬的，也就是变化的因素只有一类。

对于出现变化因素有两类的，也就是有两个变化纬度的情况，继承实现就会比较痛苦。比如上面的示例，就有两个变化纬度，一个是消息的类别，不同的消息类别处理不同；另外一个是消息的发送方式。

从理论上来说，如果用继承的方式来实现这种有两个变化纬度的情况，最后实际的实现类应该是两个纬度上可变数量的乘积那么多个。比如上面的示例，在消息类别的纬度上，目前的可变数量是 3 个，普通消息、加急消息和特急消息；在消息发送方式的纬度上，目前的可变数量也是 3 个，站内短消息、E-mail 和手机短消息。这种情况下，如果要实现全的话，那么需要的实现类应该是：$3 \times 3 = 9$ 个。

如果要在任何一个纬度上进行扩展，都需要实现另外一个纬度上的可变数量那么多个实现类，这也是为何会感觉扩展起来很困难。而且随着程序规模的加大，会越来越难以扩展和维护。

提示　**而桥接模式就是用来解决这种有两个变化纬度的情况下，如何灵活地扩展功能的一个很好的方案。其实，桥接模式主要是把继承改成了使用对象组合，从而把两个纬度分开，让每一个纬度单独去变化，最后通过对象组合的方式，把两个纬度组合起来，每一种组合的方式就相当于原来继承中的一种实现，这样就有效地减少了实际实现的类的个数。**

从理论上来说，如果用桥接模式的方式来实现这种有两个变化纬度的情况，最后实际的实现类应该是两个纬度上可变数量的和那么多个。同样是上面那个示例，使用桥接模式来实现，实现全的话，最后需要的实现类的数目应该是：$3 + 3 = 6$ 个。

这也从侧面体现了，使用对象组合的方式比继承要来得更灵活。

8. 桥接模式的调用顺序示意图

桥接模式的调用顺序如图 24.8 所示。



图 24.8　桥接模式的调用顺序示意图

24.3.2 谁来桥接

所谓谁来桥接，就是谁来负责创建抽象部分和实现部分的关系，说得更直白点，就是谁来负责创建 Implementor 对象，并把它设置到抽象部分的对象中去，这点对于使用桥接模式来说，是十分重要的一点。

大致有如下几种实现方式。

- 由客户端负责创建 Implementor 对象，并在创建抽象部分对象的时候，把它设置到抽象部分的对象中去，前面的示例采用的就是这个方式。
- 可以在抽象部分对象构建的时候，由抽象部分的对象自己来创建相应的 Implementor 对象，当然可以给它传递一些参数，它可以根据参数来选择并创建具体的 Implementor 对象。
- 可以在 Abstraction 中选择并创建一个默认的 Implementor 对象，然后子类可以根据需要改变这个实现。
- 也可以使用抽象工厂或者简单工厂来选择并创建具体的 Implementor 对象，抽象部分的类可以通过调用工厂的方法来获取 Implementor 对象。
- 如果使用 IoC/DI 容器的话，还可以通过 IoC/DI 容器来创建具体的 Implementor 对象，并注入回到 Abstraction 中。

下面分别给出后面几种实现方式的示例。

1. 由抽象部分的对象自己来创建相应的 Implementor 对象

对于这种情况的实现，又分成两种，一种是需要外部传入参数，另一种是不需要外部传入参数。

（1）从外部传递参数比较简单，比如前面的示例，如果用一个 type 来标识具体采用哪种发送消息的方案，然后在 Abstraction 的构造方法中，根据 type 进行创建就可以了。

还是用代码示例一下，主要修改 Abstraction 的构造方法。示例代码如下：

```
/**
 * 抽象的消息对象
 */
public abstract class AbstractMessage {
    /**
     * 持有一个实现部分的对象
     */
    protected MessageImplementor impl;
    /**
     * 构造方法，传入选择实现部分的类型
     * @param type 传入选择实现部分的类型
     */
    public AbstractMessage(int type){
        if(type==1){
```

```
            this.impl = new MessageSMS();
        }else if(type==2){
            this.impl = new MessageEmail();
        }else if(type==3){
            this.impl = new MessageMobile();
        }
    }
    /**
     * 发送消息，转调实现部分的方法
     * @param message 要发送的消息内容
     * @param toUser 把消息发送的目的人员
     */
    public void sendMessage(String message,String toUser){
        this.impl.send(message, toUser);
    }
}
```

（2）对于不需要外部传入参数的情况，那就说明是在 Abstraction 的实现中，根据具体的参数数据来选择相应的 Implementor 对象。有可能在 Abstraction 的构造方法中选择，也有可能在具体的方法中选择。

比如前面的示例，如果发送的消息长度在 100 以内则采用手机短消息；长度在 100～1000 以内则采用站内短消息；长度在 1000 以上采用 E-mail，那么就可以在内部方法中自己判断实现。

实现中，大致有如下改变。

- 原来 protected 的 MessageImplementor 类型的属性，不需要了，去掉。
- 提供一个 protected 的方法来获取要使用的实现部分的对象，在这个方法中，根据消息的长度来选择合适的实现对象。
- 构造方法什么都不用做了，也不需要传入参数。
- 在原来使用 impl 属性的地方，要修改成通过上面那个方法来获取合适的实现对象了，不能直接使用 impl 属性，否则会没有值。

示例代码如下：

```
public abstract class AbstractMessage {
    /**
     * 构造方法
     */
    public AbstractMessage(){
        //现在什么都不做了
    }
    /**
     * 发送消息，转调实现部分的方法
```

```
     * @param message 要发送的消息内容
     * @param toUser 把消息发送的目的人员
     */
    public void sendMessage(String message,String toUser){
        this.getImpl(message).send(message, toUser);
    }
    /**
     * 根据消息的长度来选择合适的实现
     * @param message 要发送的消息
     * @return 合适的实现对象
     */
    protected MessageImplementor getImpl(String message) {
        MessageImplementor impl = null;
        if(message == null){
            //如果没有消息，默认使用站内短消息
            impl = new MessageSMS();
        }else if(message.length()< 100){
            //如果消息长度在100以内，使用手机短消息
            impl = new MessageMobile();
        }else if(message.length()<1000){
            //如果消息长度在100~1000以内，使用站内短消息
            impl = new MessageSMS();
        }else{
            //如果消息长度在1000以上
            impl = new MessageEmail();
        }
        return impl;
    }
}
```

（3）小结

对于由抽象部分的对象自己来创建相应的 Implementor 对象的这种情况，不管是否需要外部传入参数，**优点是**客户使用简单，而且集中控制 Implementor 对象的创建和切换逻辑；**缺点是**要求 Abstraction 知道所有的具体的 Implementor 实现，并知道如何选择和创建它们，如果今后要扩展 Implementor 的实现，就要求同时修改 Abstraction 的实现，这会很不灵活，使扩展不方便。

2. 在 Abstraction 中创建默认的 Implementor 对象

对于这种方式，实现比较简单，直接在 Abstraction 的构造方法中，创建一个默认的 Implementor 对象，然后子类根据需要，看是直接使用还是覆盖掉。示例代码如下：

```
public abstract class AbstractMessage {
    protected MessageImplementor impl;
    /**
     * 构造方法
     */
    public AbstractMessage(){
        //创建一个默认的实现
        this.impl = new MessageSMS();
    }
    public void sendMessage(String message,String toUser){
        this.impl.send(message, toUser);
    }
}
```

这种方式其实还可以使用工厂方法，把创建工作延迟到子类。

3. 使用抽象工厂或者是简单工厂

对于这种方式，根据具体的需要来选择，如果是想要创建一系列实现对象，那就使用抽象工厂；如果是创建单个的实现对象，使用简单工厂就可以了。

直接在原来创建 Implementor 对象的地方，直接调用相应的抽象工厂或者是简单工厂来获取相应的 Implementor 对象。很简单，这个就不去示例了。

这种方法的优点是 Abstraction 类不用和任何一个 Implementor 类直接耦合。

4. 使用 IoC/DI 的方式

对于这种方式，Abstraction 的实现就更简单了，只需要实现注入 Implementor 对象的方法就可以了，其他的 Abstraction 就不管了。

IoC/DI 容器会负责创建 Implementor 对象，并设置回到 Abstraction 对象中，使用 IoC/DI 的方式，并不会改变 Abstraction 和 Implementor 的关系，Abstraction 同样需要持有相应的 Implementor 对象，同样会把功能委托给 Implementor 对象去实现。

24.3.3　典型例子——JDBC

在 Java 应用中，对于桥接模式有一个非常典型的例子，就是应用程序使用 JDBC 驱动程序进行开发的方式。所谓驱动程序，指的是按照预先约定好的接口来操作计算机系统或者是外围设备的程序。

先简单地回忆一下使用 JDBC 进行开发的过程。简单的片段代码示例如下：

```
        String sql = "具体要操作的 sql 语句";
        // 1: 装载驱动
        Class.forName("驱动的名字");
        // 2: 创建连接
        Connection conn = DriverManager.getConnection(
```

```
                                "连接数据库服务的URL", "用户名","密码");

    // 3: 创建 statement 或者是 preparedStatement
    PreparedStatement pstmt = conn.prepareStatement(sql);
    // 4: 执行 sql，如果是查询，再获取 ResultSet
    ResultSet rs = pstmt.executeQuery(sql);

    // 5: 循环从 ResultSet 中把值取出来，封装到数据对象中去
    while (rs.next()) {
        // 取值示意，按名称取值
        String uuid = rs.getString("uuid");
        // 取值示意，按索引取值
        int age = rs.getInt(2);
    }
    //6: 关闭
    rs.close();
    pstmt.close();
    conn.close();
```

从上面的示例可以看出，我们写的应用程序，是面向 JDBC 的 API 开发，这些接口就相当于桥接模式中抽象部分的接口。那么怎样得到这些 API 呢？是通过 DriverManager 来得到的。此时的系统结构如图 24.9 所示。

那么这些 JDBC 的 API，谁去实现呢？光有接口，没有实现也不行啊。

该驱动程序登场了，JDBC 的驱动程序实现了 JDBC 的 API，驱动程序就相当于桥接模式中的具体实现部分。而且不同的数据库，由于数据库实现不一样，可执行的 sql 也不完全一样，因此对于 JDBC 驱动的实现也是不一样的，也就是不同的数据库会有不同的驱动实现。此时驱动程序的程序结构如图 24.10 所示。

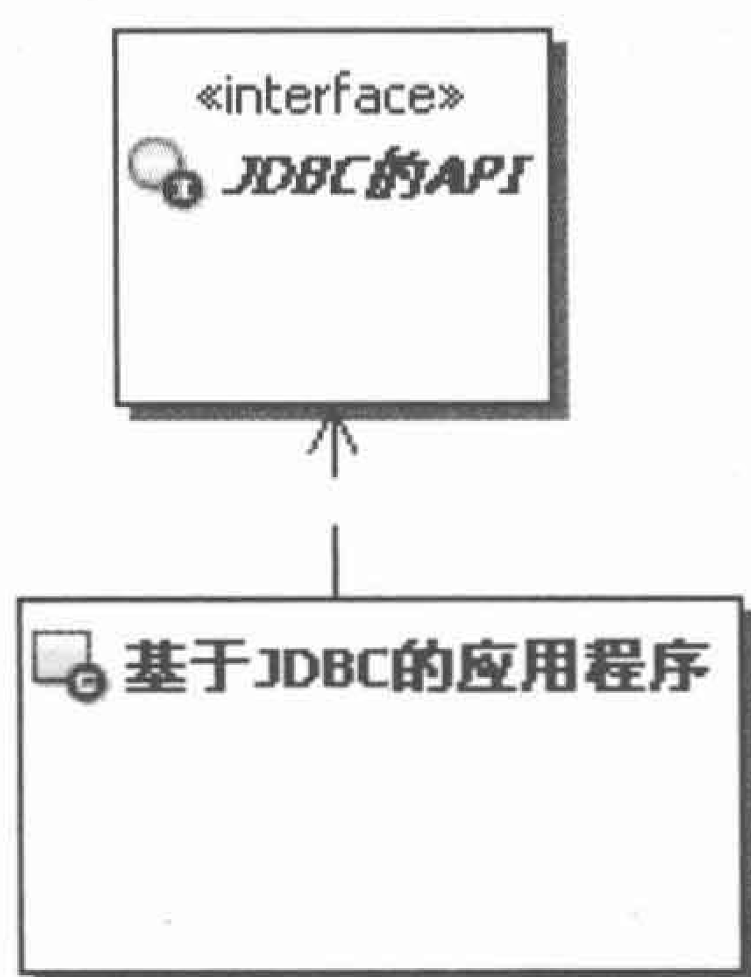


图 24.9　基于 JDBC 开发的应用程序结构示意图

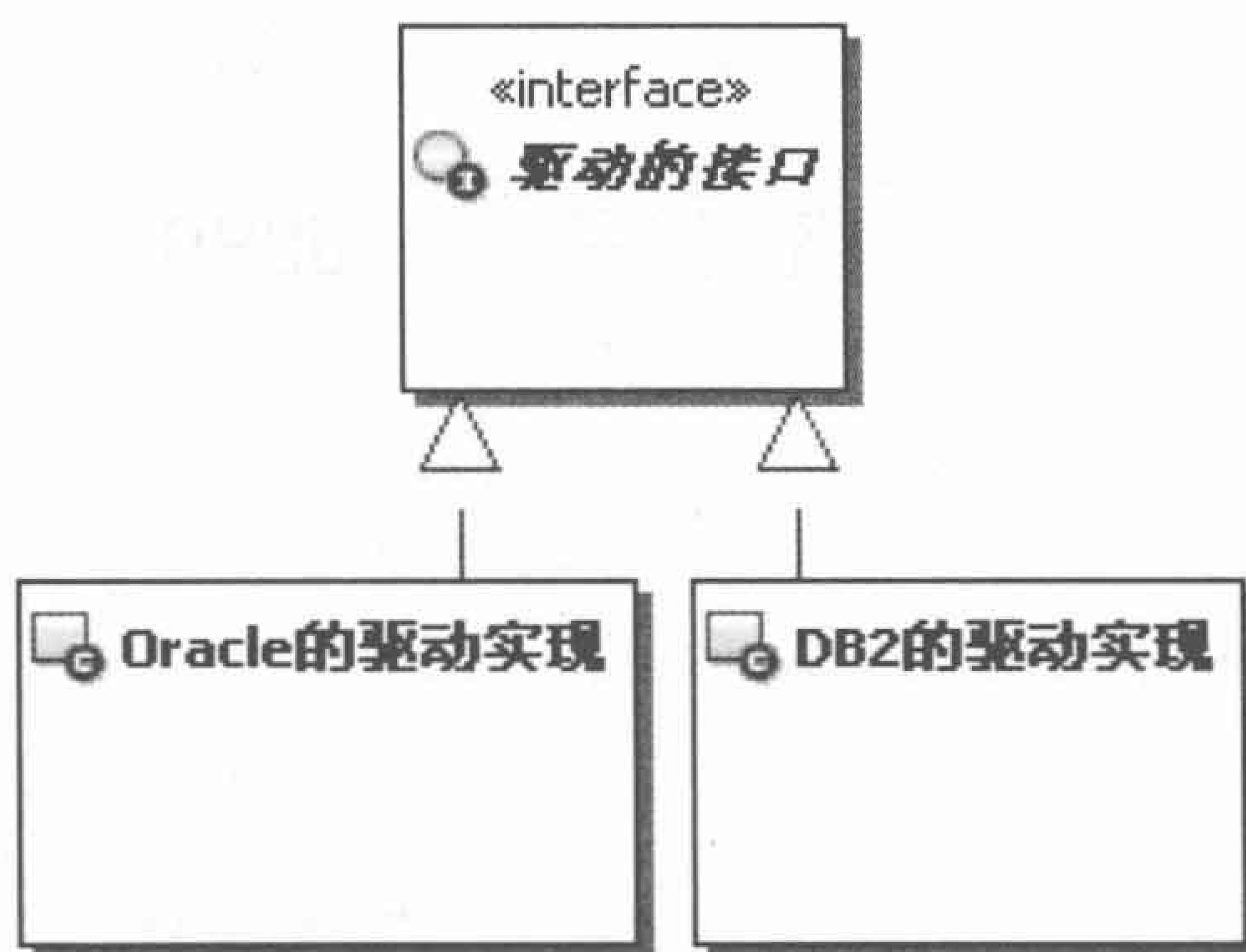


图 24.10　驱动程序实现结构示意图

有了抽象部分——JDBC 的 API 和具体实现部分——驱动程序，那么它们如何连接起来呢？就是如何桥接呢？

就是前面提到的 DriverManager 来把它们桥接起来。从某个侧面来看，DriverManager

在这里起到了类似于简单工厂的功能，基于 JDBC 的应用程序需要使用 JDBC 的 API，如何得到呢？就通过 DriverManager 来获取相应的对象。

那么此时系统的整体结构如图 24.11 所示。

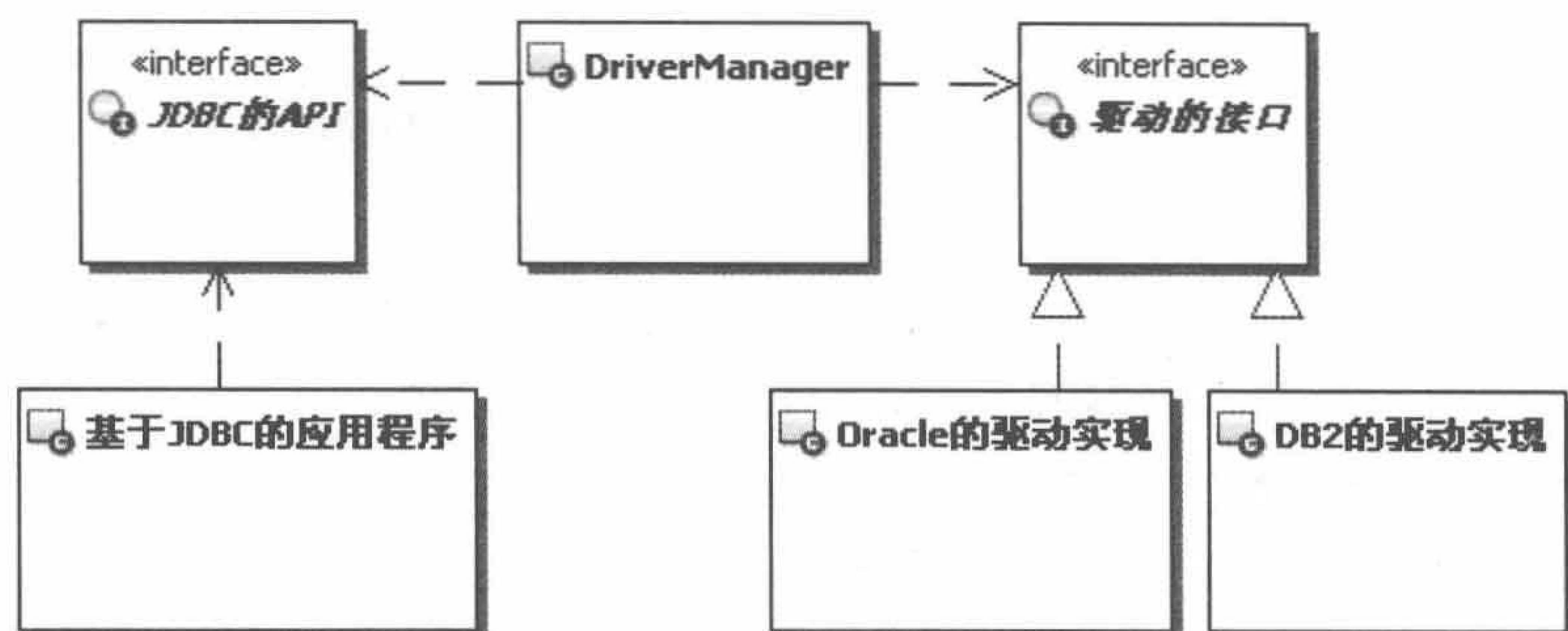


图 24.11　JDBC 的结构示意图

通过图 24.11 可以看出，基于 JDBC 的应用程序，使用 JDBC 的 API，相当于是对数据库操作的抽象扩展，算做桥接模式的抽象部分；而具体的接口实现是由驱动来完成的，驱动就相当于桥接模式的实现部分了。而桥接的方式，不再是让抽象部分持有实现部分，而是采用了类似于工厂的做法，通过 DriverManager 来把抽象部分和实现部分对接起来，从而实现抽象部分和实现部分解耦。

JDBC 的这种架构，把抽象部分和具体部分分离开来，从而使得抽象部分和具体部分都可以独立扩展。对于应用程序而言，只要选用不同的驱动，就可以让程序操作不同的数据库，而无需更改应用程序，从而实现在不同的数据库上移植；对于驱动程序而言，为数据库实现不同的驱动程序，并不会影响应用程序。而且，JDBC 的这种架构，还合理地划分了应用程序开发人员和驱动程序开发人员的边界。

延伸

对于有些朋友会认为，从局部来看，体现了策略模式，比如，在上面的结构中删除“JDBC 的 API 和基于 JDBC 的应用程序”，那么剩下的部分，看起来就是一个策略模式的体现。此时的 DriverManager 相当于上下文，而各个具体驱动的实现就相当于是具体的策略实现。这个理解也不算错，但是在这里看来，这样理解是比较片面的。

对于这个问题，再次强调一点：对于设计模式，要从整体结构上、本质目标上和思想体现上来把握，而不要从局部、表现和特例实现上来把握。

24.3.4　广义桥接——Java 中无处不桥接

使用 Java 编写程序，一个很重要的原则就是“面向接口编程”，说得准确点应该是“面向抽象编程”，由于在 Java 开发中，更多地使用接口而非抽象类，因此通常就说成“面向接口编程”了。

接口把具体的实现和使用接口的客户程序分离开来，从而使得具体的实现和使用接口的客户程序可以分别扩展，而不会相互影响。使用接口的程序结构如图 24.12 所示。

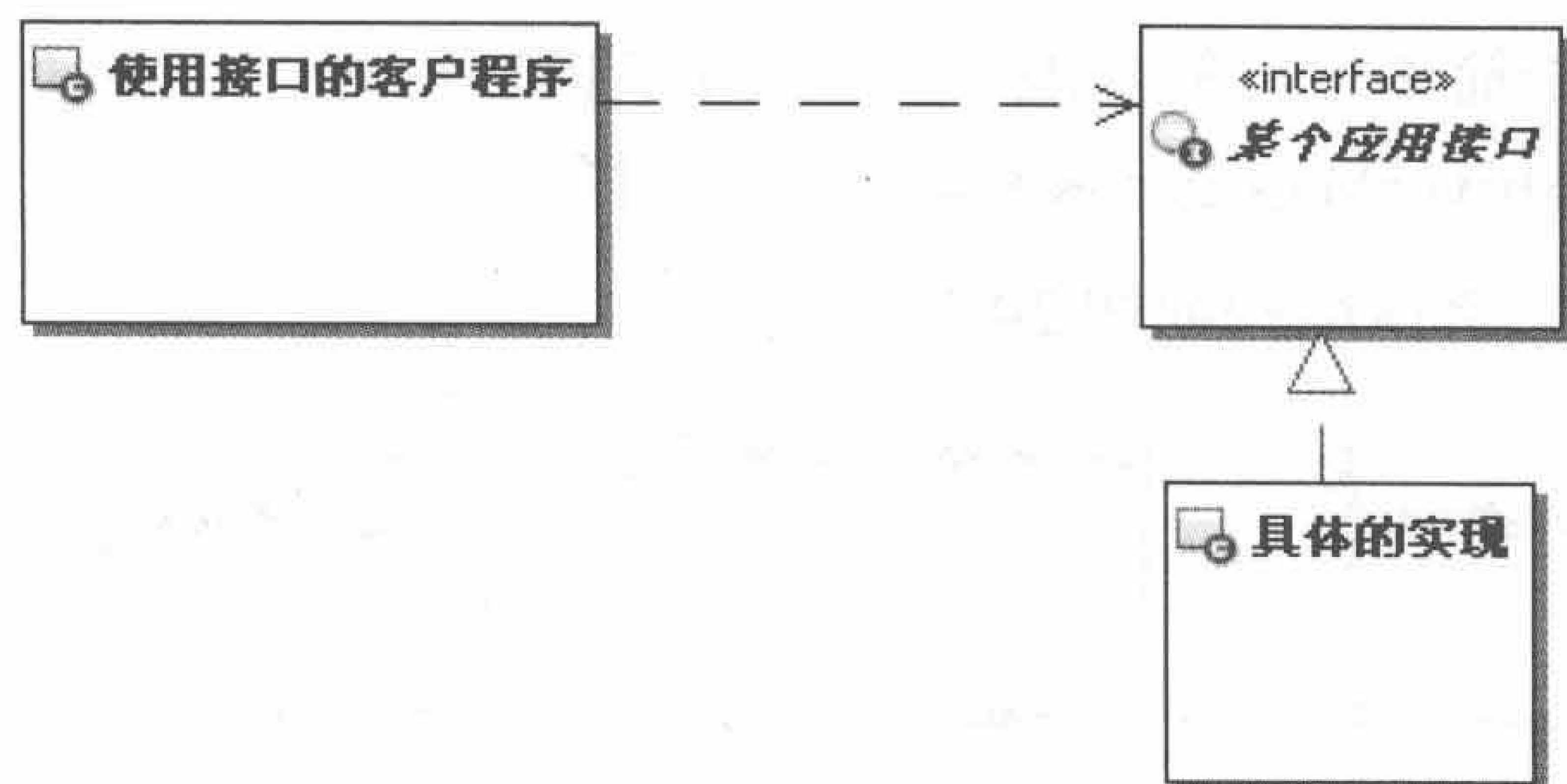


图 24.12　使用接口的程序结构示意图

可能有些朋友会觉得，听起来怎么像是桥接模式的功能呢？没错，如果把桥接模式的抽象部分先稍稍简化一下，暂时不要 RefinedAbstraction 部分，那么就跟图 24.12 所示的结构图差不多了。删除 RefinedAbstraction 后的简化的桥接模式结构示意图如图 24.13 所示。

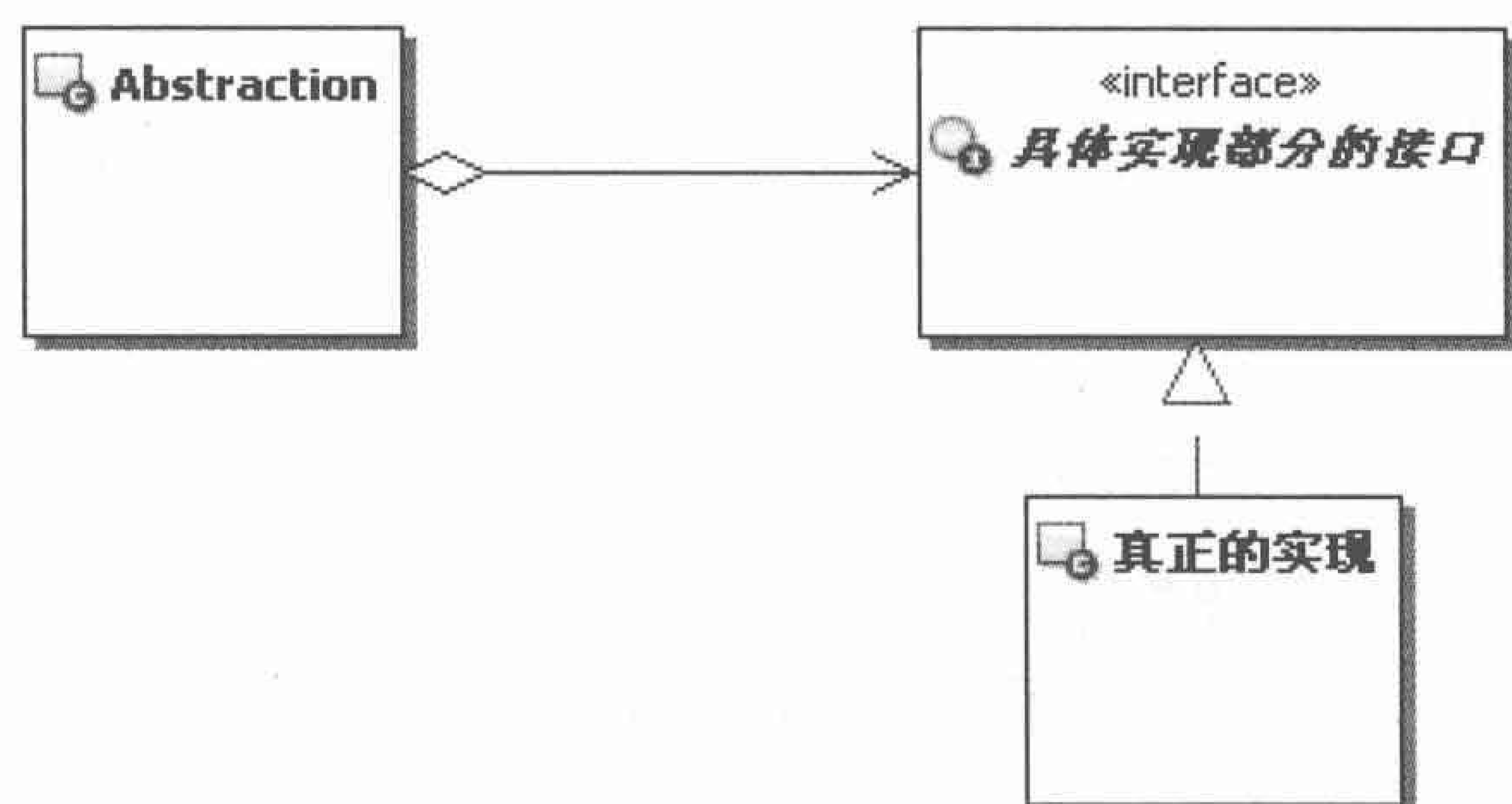


图 24.13　简化的桥接模式结构示意图

是不是差不多呢？有朋友可能会觉得还是有很大差异，差异主要在前面接口的客户程序是直接使用接口对象，不像桥接模式的抽象部分那样是持有具体实现部分的接口，这就导致画出来的结构图一个是依赖，一个是聚合关联。

请思考它们的本质功能，桥接模式中的抽象部分持有具体实现部分的接口，最终目的是什么？是为了通过调用具体实现部分的接口中的方法来完成一定的功能，这和直接使用接口差不多，只是表现形式有点不一样。再说，前面那个使用接口的客户程序也可以持有相应的接口对象，这样从形式上就一样了。

也就是说，**从某个角度来讲，桥接模式就是对“面向抽象编程”设计原则的扩展。**正是通过具体实现的接口，把抽象部分和具体的实现部分分离开来，抽象部分相当于是使用实现部分接口的客户程序。这样抽象部分和实现部分就松散耦合了，从而可以实现相互独立的变化。

这样一来，几乎可以把所有面向抽象编写的程序，都视作是桥接模式的体现，至少也是简化的桥接模式，就算是广义的桥接吧。而 Java 编程很强调“面向抽象编程”，因此，广义的桥接，在 Java 中可以说是无处不在。

再举个大家最熟悉的例子来示例一下。在 Java 应用开发中，分层实现是最基本的设计方式，就拿大家最熟悉的三层架构表现层、逻辑层和数据层来说，或许有些朋友对它们称呼的名称不同，但都是同一回事情。

三层的基本关系是表现层调用逻辑层，逻辑层调用数据层。通过什么方式来进行调用呢？当然是接口了，它们的基本结构如图 24.14 所示。

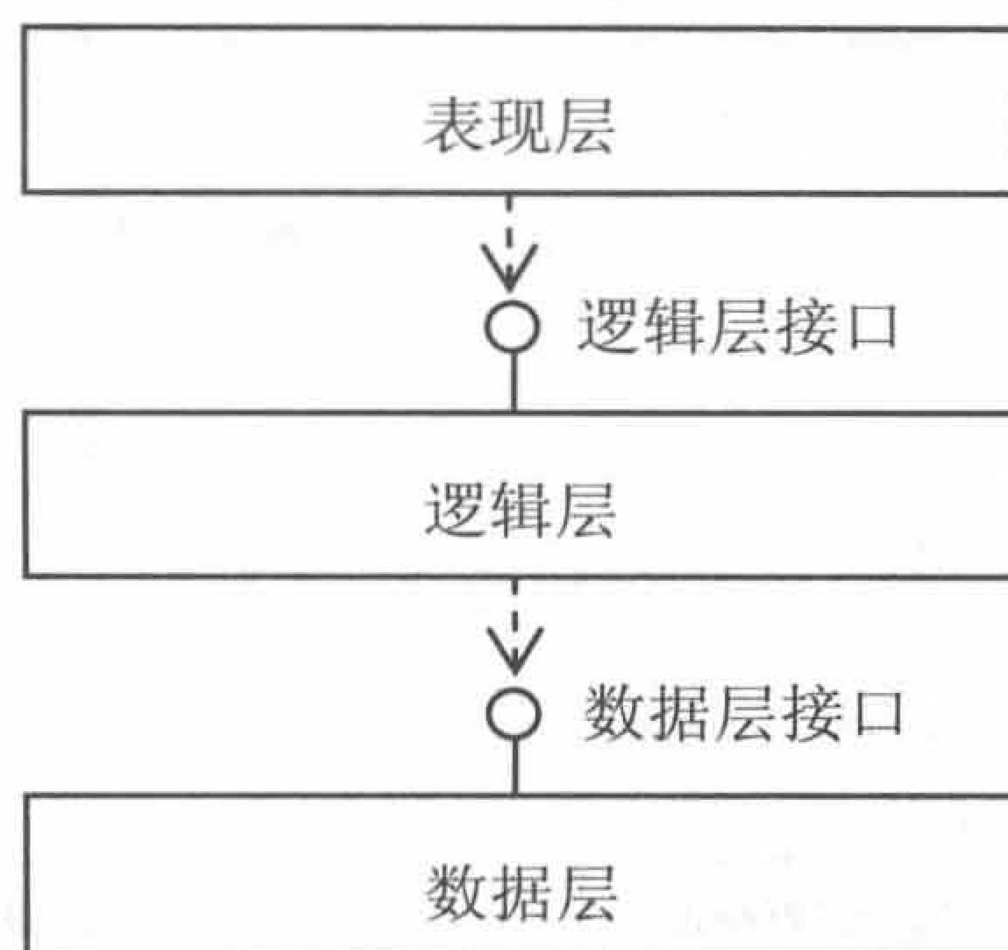


图 24.14　基本的三层架构示意图

通过接口来进行调用，使得表现层和逻辑层分离开来，也就是说表现层的变化不会影响到逻辑层，同理逻辑层的变化也不会影响到表现层。这也是同一套逻辑层和数据层可以同时支持不同的表现层实现的原因，比如支持 Swing 或 Web 方式的表现层。

在逻辑层和数据层之间也是通过接口来调用，同样将逻辑层和数据层分离，使得它们可以独立地扩展。尤其是数据层，可能会有很多的实现方式，比如数据库实现、文件实现等，即使是数据库实现，又有针对不同数据库的实现，如 Oracle、DB2 等。

总之，通过面向抽象编程，三层架构的各层都能够独立地扩展和变化，而不会对其他层产生影响。这正是桥接模式的功能，实现抽象和实现的分离，从而使得它们可以独立地变化。当然三层架构不只是在一个地方使用桥接模式，而是至少在两个地方来使用桥接模式，一个是在表现层和逻辑层之间，一个是在逻辑层和数据层之间。

下面先来分别看看这两个使用桥接模式地方的程序结构，然后再综合起来看看整体的程序结构。

逻辑层和数据层之间的程序结构如图 24.15 所示。

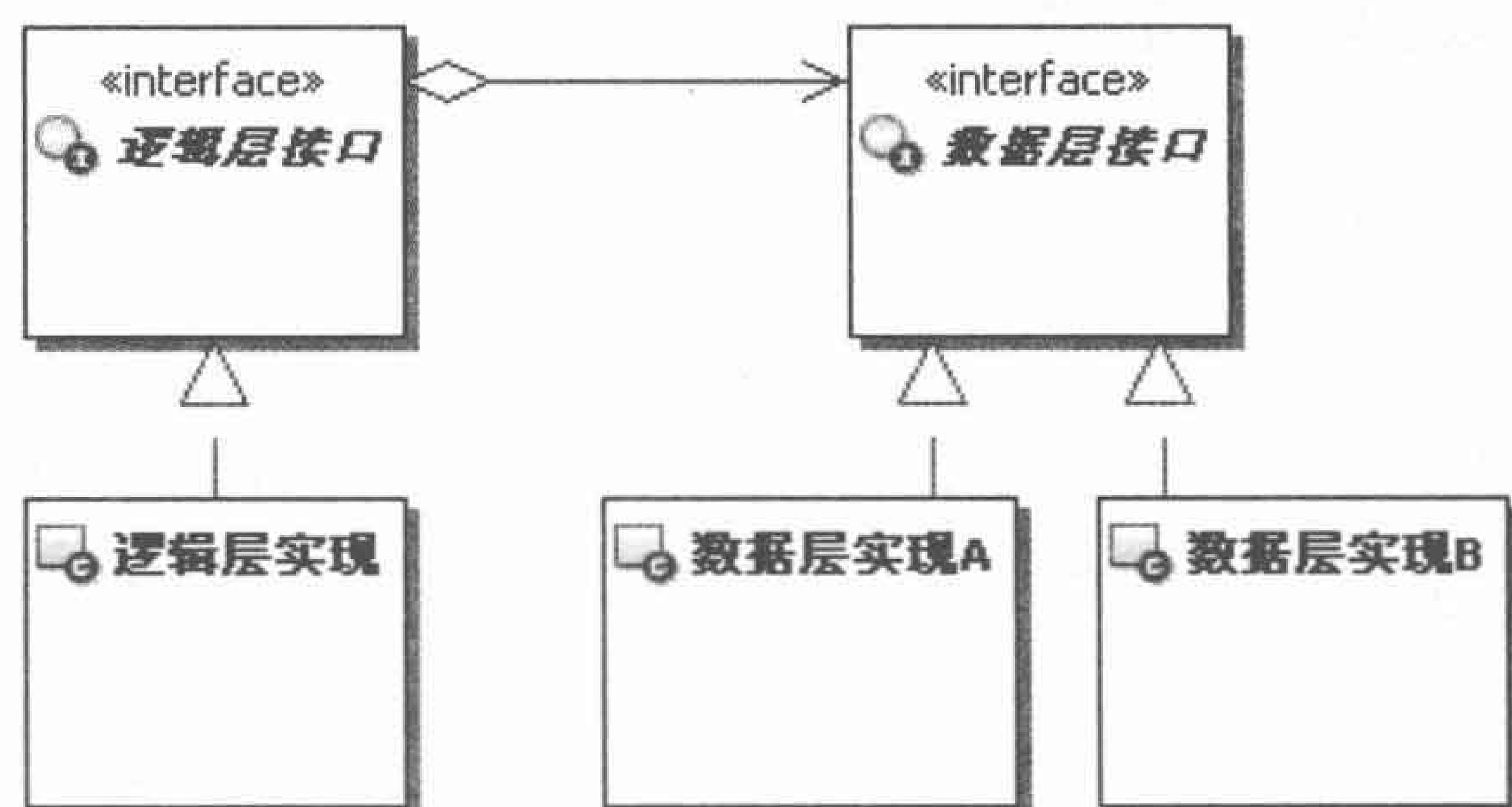


图 24.15　逻辑层和数据层的程序结构示意图

表现层和逻辑层之间的结构示意如图 24.16 所示。

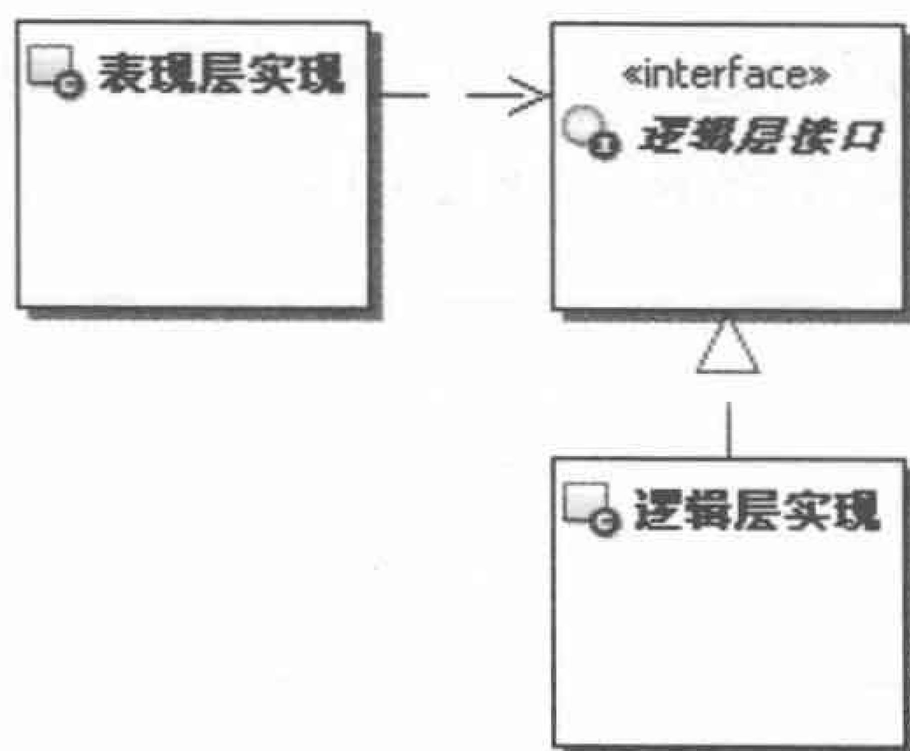


图 24.16　表现层和逻辑层的结构示意图

把逻辑层、数据层和表现层结合起来。结合后的程序结构如图 24.17 所示。

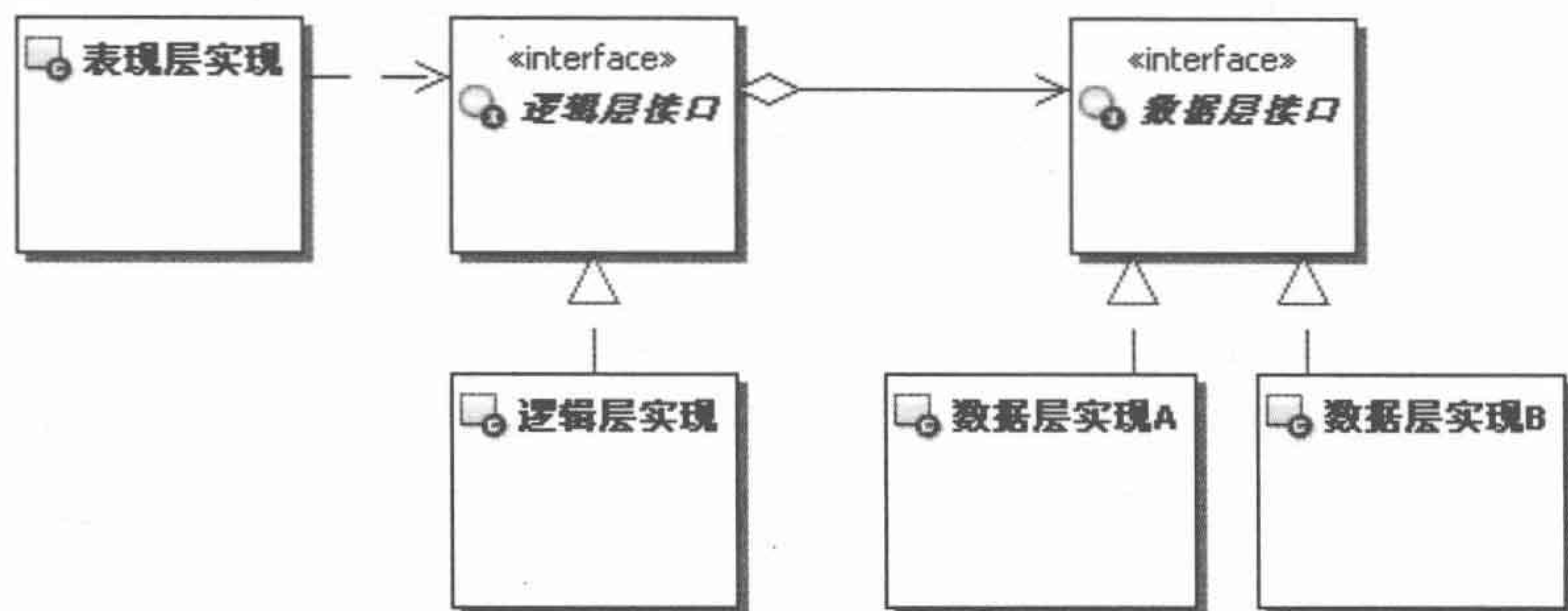


图 24.17　三层结合的结构示意图

从广义桥接模式的角度来看，平日熟悉的三层架构其实就是在组合使用桥接模式。从这个图还可以看出，**桥接模式是可以连续组合使用的，一个桥接模式的实现部分，可以作为下一个桥接模式的抽象部分**。以此类推，可以从三层架构扩展到四层、五层，直到 *N* 层架构，都可以使用桥接模式来组合。

如果从更本质的角度来看，基本上只要是面向抽象编写的 Java 程序都可以视为是桥接模式的应用，都是让抽象和实现相分离，从而使它们能独立地变化。不过只要分离的目的达到了，叫不叫桥接模式则无所谓了。

24.3.5　桥接模式的优点

- 分离抽象和实现部分

  桥接模式分离了抽象部分和实现部分，从而极大地提高了系统的灵活性。让抽象部分和实现部分独立开来，分别定义接口，这有助于对系统进行分层，从而产生更好的结构化的系统。对于系统的高层部分，只需要知道抽象部分和实现部分的接口就可以了。

- 更好的扩展性

  由于桥接模式把抽象部分和实现部分分离开了，而且分别定义接口，这就使得抽象部分和实现部分可以分别独立地扩展，而不会相互影响，从而大大地提高了系统的可扩展性。

- 可动态地切换实现

由于桥接模式把抽象部分和实现部分分离开了，所以在实现桥接的时候，就可以实现动态的选择和使用具体的实现。也就是说一个实现不再是固定的绑定在一个抽象接口上了，可以实现运行期间动态地切换。

- 可减少子类的个数

  根据前面的讲述，对于有两个变化纬度的情况，如果采用继承的实现方式，大约需要两个纬度上的可变化数量的乘积个子类；而采用桥接模式来实现，大约需要两个纬度上的可变化数量的和个子类。可以明显地减少子类的个数。

24.3.6　思考桥接模式

1. 桥接模式的本质

> 桥接模式的本质：**分离抽象和实现**。

桥接模式最重要的工作就是分离抽象部分和实现部分，这是解决问题的关键。只有把抽象部分和实现部分分离开了，才能够让它们独立地变化；只有抽象部分和实现部分可以独立地变化，系统才会有更好的可扩展性和可维护性。

还有其他的好处，比如，可以动态地切换实现、可以减少子类个数等。都是把抽象部分和实现部分分离以后带来的。如果不把抽象部分和实现部分分离开，那一切就无从谈起。所以综合来说，桥接模式的本质在于“分离抽象和实现”。

2. 对设计原则的体现

（1）桥接模式很好地实现了开闭原则。

通常应用桥接模式的地方，抽象部分和实现部分都是可变化的，也就是应用会有两个变化纬度，桥接模式就是找到这两个变化，并分别封装起来，从而合理地实现 OCP。

在使用桥接模式的时候，通常情况下，顶层的 Abstraction 和 Implementor 是不变的，而具体的 Implementor 的实现是可变的。由于 Abstraction 是通过接口来操作具体的实现，因此具体的 Implementor 的实现是可以扩展的，根据需要可以有多个具体的实现。

同样地，RefinedAbstraction 也是可变的，它继承并扩展 Abstraction，通常在 RefinedAbstraction 的实现中，会调用 Abstraction 中的方法，通过组合使用来完成更多的功能，这些功能常常是与具体业务有关系的。

（2）桥接模式还很好地体现了：多用对象组合，少用对象继承。

在前面的示例中，如果使用对象继承来扩展功能，不但让对象之间有很强的耦合性，而且会需要很多的子类才能够完成相应的功能，前面已经讲述过了，需要两个纬度上的可变化数量的乘积个子类。

而采用对象的组合，松散了对象之间的耦合性，不但使每个对象变得简单和可维护，还大大减少了子类的个数，根据前面的讲述，大约需要两个纬度上的可变化数量的和这么多个子类。

3. 何时选用桥接模式

建议在以下情况中选用桥接模式。

- 如果你不希望在抽象部分和实现部分采用固定的绑定关系，可以采用桥接模式，来把抽象部分和实现部分分开，然后在程序运行期间来动态地设置抽象部分需要用到的具体的实现，还可以动态地切换具体的实现。
- 如果出现抽象部分和实现部分都能够扩展的情况，可以采用桥接模式，让抽象部分和实现部分独立地变化，从而灵活地进行单独扩展，而不是搅在一起，扩展一边就会影响到另一边。
- 如果希望实现部分的修改不会对客户产生影响，可以采用桥接模式。由于客户是面向抽象的接口在运行，实现部分的修改可以独立于抽象部分，并不会对客户产生影响，也可以说对客户是透明的。
- 如果采用继承的实现方案，会导致产生很多子类，对于这种情况，可以考虑采用桥接模式，分析功能变化的原因，看看是否能分离成不同的纬度，然后通过桥接模式来分离它们，从而减少子类的数目。

24.3.7 相关模式

- 桥接模式和策略模式

  这两个模式有很大的相似之处。

  如果把桥接模式的抽象部分简化来看，暂时不去扩展 Abstraction，也就是去掉 RefinedAbstraction。桥接模式简化后的结构图如图 24.13 所示。再看看策略模式的结构图，参见图 17.1。会发现，这个时候它们的结构都类似图 24.18 所示。

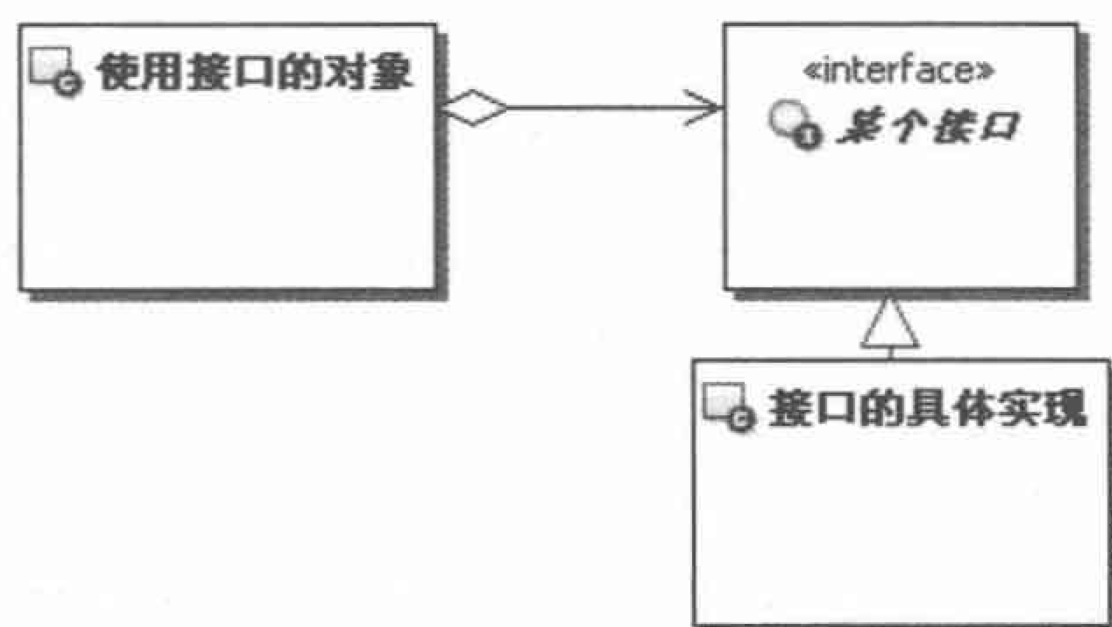


图 24.18 桥接模式和策略模式结构示意图

  通过上面的结构图，可以体会到桥接模式和策略模式是如此相似。可以把策略模式的 Context 当做是使用接口的对象，而 Strategy 就是某个接口了，具体的策略实现就相当于接口的具体实现。这样看来的话，某些情况下，可以使用桥接模式来模拟实现策略模式的功能。

  这两个模式虽然相似，但也还是有区别的。最主要的是模式的目的不一样，策略模式的目的是封装一系列的算法，使得这些算法可以相互替换；而桥接模式的目的是分离抽象部分和实现部分，使得它们可以独立地变化。

- 桥接模式和状态模式

  由于从模式结构上看，状态模式和策略模式是一样的，因此这两个模式的关系

也基本上类似于桥接模式和策略模式的关系。只不过状态模式的目的是封装状态对应的行为，并在内部状态改变的时候改变对象的行为。

- 桥接模式和模板方法模式

  这两个模式有相似之处。

  虽然标准的模板方法模式是采用继承来实现的，但是模板方法也可以通过回调接口的方式来实现。如果把接口的实现独立出去，那就类似于模板方法通过接口去调用具体的实现方法了，这样的结构就和简化的桥接模式类似了。

  可以使用桥接模式来模拟实现模板方法模式的功能。如果在实现 Abstraction 对象的时候，在其中定义方法，方法中就是某个固定的算法骨架，也就是说这个方法就相当于模板方法。在模板方法模式中，是把不能确定实现的步骤延迟到子类去实现；现在在桥接模式中，把不能确定实现的步骤委托给具体实现部分去完成，通过回调实现部分的接口，来完成算法骨架中的某些步骤。这样一来，就可以实现使用桥接模式来模拟实现模板方法模式的功能。

  使用桥接模式来模拟实现模板方法模式的功能，还有一个潜在的好处，就是模板方法也可以很方便地扩展和变化。在标准的模板方法中，一个问题就是当模板发生变化的时候，所有的子类都要变化，非常不方便。而使用桥接模式来实现类似的功能，就没有这个问题。

> 注意　**另外，这里只是说从实现具体的业务功能上，桥接模式可以模拟实现模板方法模式能实现的功能，并不是说桥接模式和模板方法模式就变成一样的，或者是桥接模式就可以替换模板方法模式了。要注意它们本身的功能、目的、本质思想都是不一样的。**

- 桥接模式和抽象工厂模式

  这两个模式可以组合使用。

  桥接模式中，抽象部分需要获取相应的实现部分的接口对象，那么谁来创建实现部分的具体实现对象呢？这就是抽象工厂模式派上用场的地方。也就是使用抽象工厂模式来创建和配置一个特定的具体的实现对象。

  事实上，抽象工厂主要是用来创建一系列对象的，如果创建的对象很少，或者是很简单，还可以采用简单工厂，也能达到同样的效果，但是会比抽象工厂来得简单。

- 桥接模式和适配器模式

  这两个模式可以组合使用。

  这两个模式功能是完全不一样的，适配器模式的功能主要是用来帮助无关的类协同工作，重点在解决原本由于接口不兼容而不能一起工作的那些类，使得它们可以一起工作。而桥接模式则重点在分离抽象部分和实现部分。

  所以在使用上，通常在系统设计完成以后，才会考虑使用适配器模式；而桥接模式是在系统开始的时候就要考虑使用。

虽然功能上不一样，这两个模式还是可以组合使用的，比如，已有实现部分的接口，但是有些不太适应现在新的功能对接口的需要，完全抛弃吧，有些功能还用得上，该怎么办呢？那就使用适配器来进行适配，使得旧的接口能够适应新的功能的需要。

第 25 章　访问者模式（Visitor）

25.1 场景问题

25.1.1 扩展客户管理的功能

考虑这样一个应用：扩展客户管理的功能。

既然是扩展功能，那么肯定是已经存在一定的功能了，先看看已有的功能，公司的客户分成两大类，一类是企业客户，一类是个人客户，现有的功能非常简单，就是能让客户提出服务申请。目前的程序结构如图 25.1 所示。

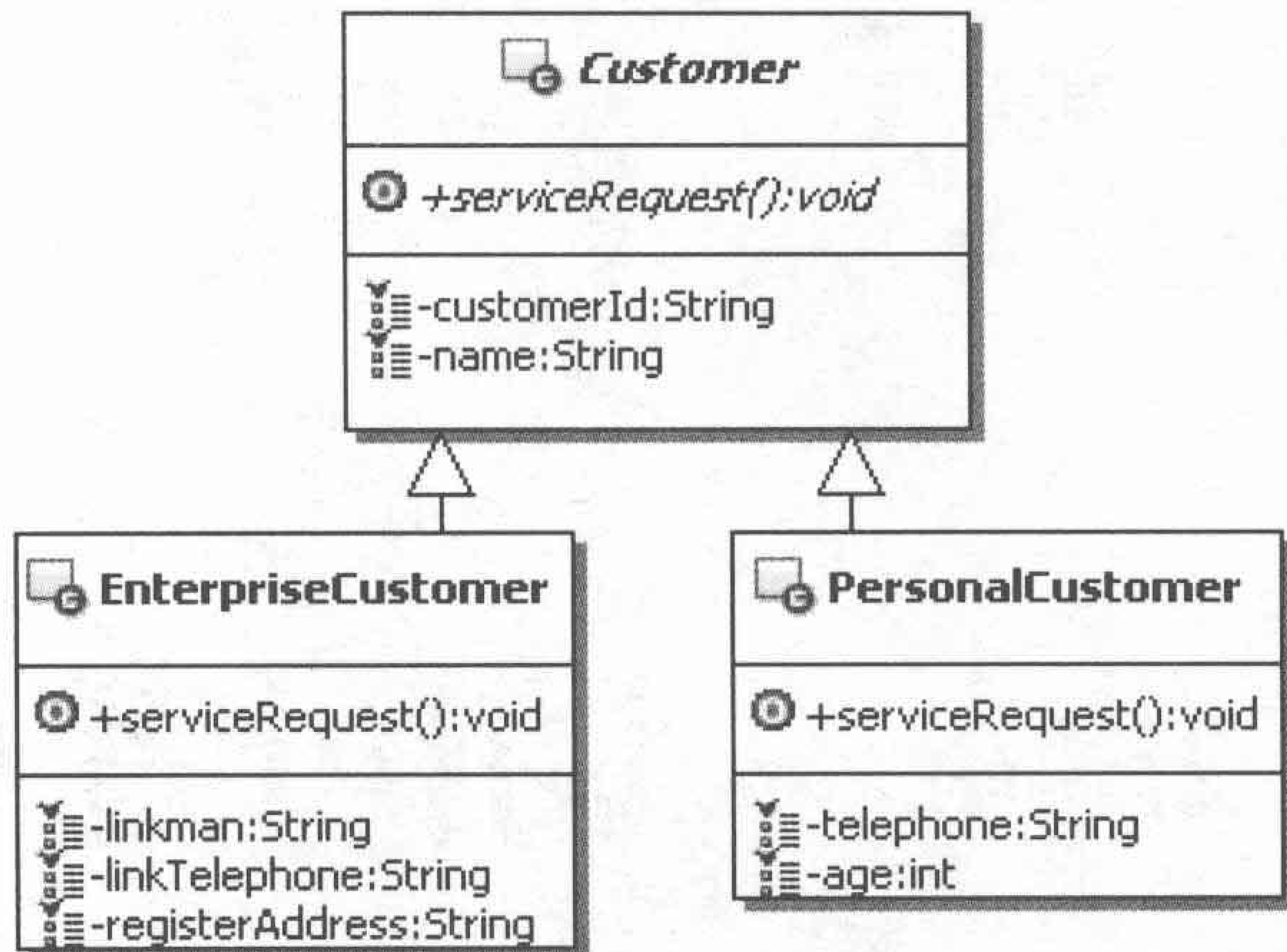


图 25.1　已有的客户管理程序结构示意图

现有的实现很简单，先看看 Customer 的实现。示例代码如下：

```
/**
 * 各种客户的父类
 */
public abstract class Customer {
    /**
     * 客户编号
     */
    private String customerId;
    /**
     * 客户名称
     */
    private String name;
```

属性对应的 getter/setter 方法，为了篇幅关系，省略了

```
/**
     * 客户提出服务请求的方法，示意一下
     */
```

```
    public abstract void serviceRequest();
}
```

接下来看看企业客户的实现。示例代码如下：

```
/**
 * 企业客户
 */
public class EnterpriseCustomer extends Customer{
    /**
     * 联系人
     */
    private String linkman;
    /**
     * 联系电话
     */
    private String linkTelephone;
    /**
     * 企业注册地址
     */
    private String registerAddress;

    /**
     * 企业客户提出服务请求的方法，示意一下
     */
    public void serviceRequest(){
        //企业客户提出的具体服务请求
        System.out.println(this.getName()+"企业提出服务请求");
    }
}
```


再来看看个人客户的实现。示例代码如下：

```
/**
 * 个人客户
 */
public class PersonalCustomer extends Customer{
    /**
     * 联系电话
     */
    private String telephone;
```

```
    /**
     * 年龄
     */
    private int age;
    /**
     * 企业注册地址
     */
    private String registerAddress;

    /**
     * 个人客户提出服务请求的方法，示意一下
     */
    public void serviceRequest(){
        //个人客户提出的具体服务请求
        System.out.println("客户"+this.getName()+"提出服务请求");
    }
}
```

属性对应的 getter/setter 方法，为了篇幅关系，省略了

从上面的实现可以看出来，以前对客户的管理功能是很少的，现在随着业务的发展，需要加强对客户管理的功能。假设现在需要增加以下功能。

- 客户对公司产品的偏好分析。针对企业客户和个人客户有不同的分析策略，主要是根据以往购买的历史、潜在购买意向等进行分析，对于企业客户还要添加上客户所在行业的发展趋势、客户的发展预期等分析。
- 客户价值分析。针对企业客户和个人客户，有不同的分析方式和策略。主要是根据购买的金额大小、购买的产品和服务的多少、购买的频率等进行分析。

其实除了这些功能，还有很多潜在的功能，只是现在还没有要求实现，比如，针对不同的客户进行需求调查；针对不同的客户进行满意度分析；客户消费预期分析等。虽然现在没有要求实现，但不排除今后有可能会要求实现。

25.1.2 不用模式的解决方案

要实现上面要求的功能，也不是很困难，一个很基本的思路就是，既然不同类型的客户操作是不同的，那么在不同类型的客户中分别实现这些功能就可以了。

由于这些功能的实现依附于很多其他功能的实现，或者是需要很多其他的业务数据，在示例中不太好完整地体现其功能实现，都是示意一下，因此提前说明一下。

按照上述的想法，这个时候的程序结构如图 25.2 所示。

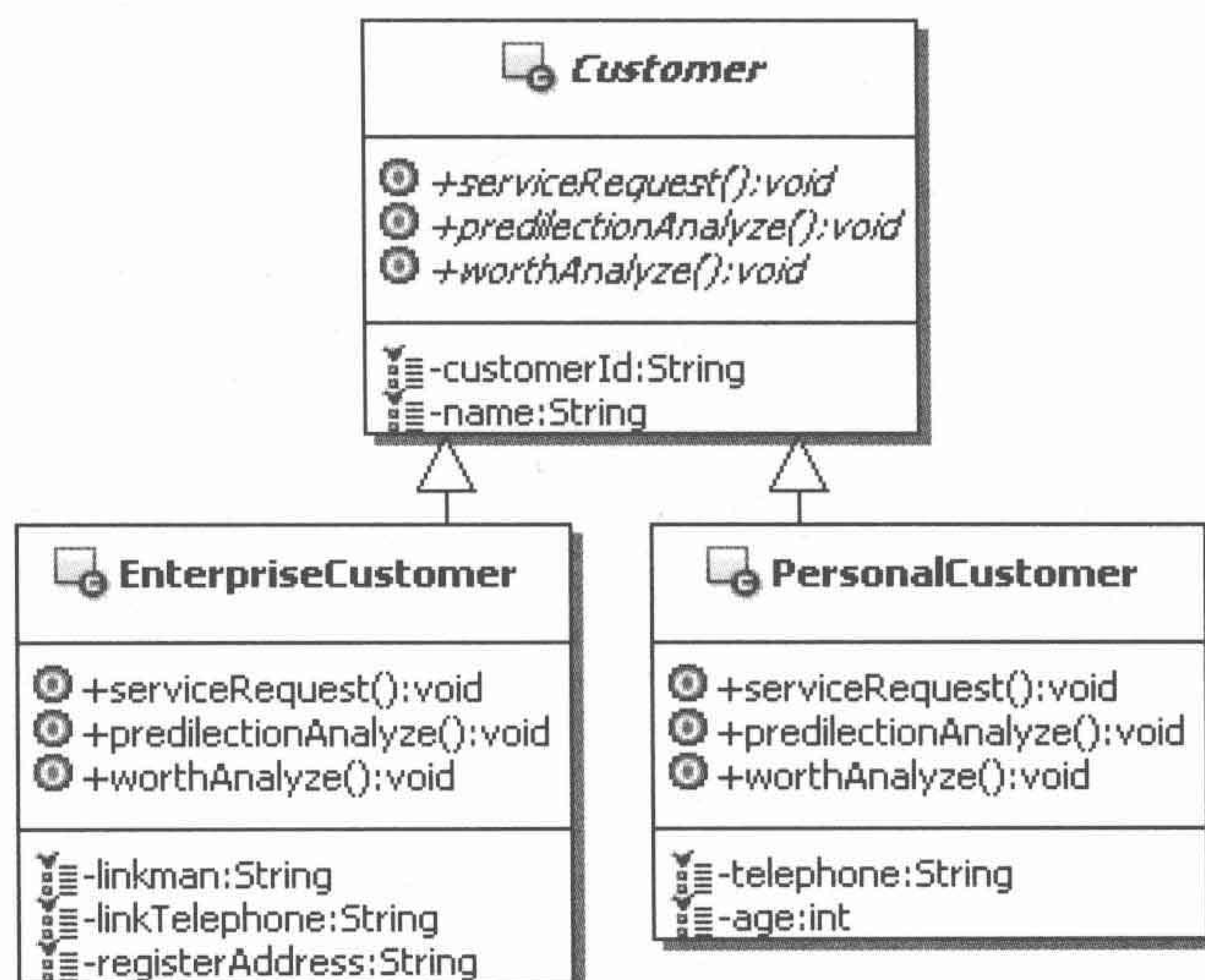


图 25.2　扩展客户管理功能的结构示意图

按照这个思路，把程序示意实现出来。

（1）先看看抽象的父类，主要就是加入了两个新的方法。示例代码如下：

```
public abstract class Customer {
    private String customerId;
    private String name;
                          属性对应的 getter/setter 方法，为
                          了篇幅关系，省略了

    public abstract void serviceRequest();

    /**
     * 客户对公司产品的偏好分析，示意一下
     */
    public abstract void predilectionAnalyze();
    /**
     * 客户价值分析，示意一下
     */
    public abstract void worthAnalyze();
}
```

（2）再来看看企业客户的示意实现。

```
public class EnterpriseCustomer extends Customer{
    private String linkman;
    private String linkTelephone;
    private String registerAddress;
```

```
    属性对应的getter/setter方法，为
    了篇幅关系，省略了

    public void serviceRequest(){
        //企业客户提出的具体服务请求
        System.out.println(this.getName()+"企业提出服务请求");
    }

    /**
     * 企业客户对公司产品的偏好分析，示意一下
     */
    public void predilectionAnalyze(){
        //根据以往购买的历史、潜在购买意向
        //以及客户所在行业的发展趋势、客户的发展预期等的分析
        System.out.println("现在对企业客户"+this.getName()
                                        +"进行产品偏好分析");
    }

    /**
     * 企业客户价值分析，示意一下
     */
    public void worthAnalyze(){
        //根据购买的金额大小、购买的产品和服务的多少、购买的频率等进行分析
        //企业客户的标准会比个人客户的高
        System.out.println("现在对企业客户"+this.getName()
                                        +"进行价值分析");
    }
}
```

（3）接下来看看个人客户的示意实现。示例代码如下：

```
public class PersonalCustomer extends Customer{
    private String telephone;
    private int age;

    属性对应的getter/setter方法，为
    了篇幅关系，省略了

    public void serviceRequest(){
        //个人客户提出的具体服务请求
        System.out.println("客户"+this.getName()+"提出服务请求");
```

```
    }

    /**
     * 个人客户对公司产品的偏好分析，示意一下
     */
    public void predilectionAnalyze(){
        System.out.println("现在对个人客户"+this.getName()
                                +"进行产品偏好分析");
    }

    /**
     * 个人客户价值分析，示意一下
     */
    public void worthAnalyze(){
        System.out.println("现在对个人客户"+this.getName()
                                +"进行价值分析");
    }
}
```

（4）如何使用上面实现的功能呢？写个客户端来测试一下。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //准备些测试数据
        Collection<Customer> colCustomer = preparedTestData();
        //循环对客户进行操作
        for(Customer cm : colCustomer){
            //进行偏好分析
            cm.predilectionAnalyze();
            //进行价值分析
            cm.worthAnalyze();
        }
    }
    private static Collection<Customer> preparedTestData(){
        Collection<Customer> colCustomer =
                            new ArrayList<Customer>();
        //为了测试方便，准备些数据
        Customer cm1 = new EnterpriseCustomer();
        cm1.setName("ABC 集团");
        colCustomer.add(cm1);

        Customer cm2 = new EnterpriseCustomer();
```

```
        cm2.setName("CDE公司");
        colCustomer.add(cm2);

        Customer cm3 = new PersonalCustomer();
        cm3.setName("张三");
        colCustomer.add(cm3);

        return colCustomer;
    }
}
```

运行结果如下：

```
现在对企业客户ABC集团进行产品偏好分析
现在对企业客户ABC集团进行价值分析
现在对企业客户CDE公司进行产品偏好分析
现在对企业客户CDE公司进行价值分析
现在对个人客户张三进行产品偏好分析
现在对个人客户张三进行价值分析
```

25.1.3 有何问题

以很简单的方式，实现了要求的功能，这种实现有没有什么问题呢？仔细分析上面的实现，发现有以下两个主要的问题。

- 在企业客户和个人客户的类中，都分别实现了提出服务请求、进行产品偏好分析、进行客户价值分析等功能，也就是说，这些功能的实现代码是混杂在同一个类中的；而且相同的功能分散到了不同的类中去实现，会导致整个系统难以理解、难以维护。
- 更为痛苦的是，采用这样的实现方式，如果要给客户扩展新的功能，比如前面提到的针对不同的客户进行需求调查、针对不同的客户进行满意度分析、客户消费预期分析等。每次扩展，都需要改动企业客户的类和个人客户的类，当然也可以通过为它们扩展子类的方式，但是这样可能会造成过多的对象层次。

问题

那么有没有办法，能够在不改变客户对象结构中各元素类的前提下，为这些类定义新的功能？也就是要求不改变企业客户和个人客户类，就能为企业客户和个人客户类定义新的功能？

25.2 解决方案

25.2.1 使用访问者模式来解决问题

用来解决上述问题的一个合理的解决方案，就是使用访问者模式。那么什么是访问者模式呢？

（1）访问者模式的定义

> 表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

（2）应用访问者模式来解决的思路

仔细分析上面的示例。对于客户这个对象结构，不想改变类，又要添加新的功能，很明显就需要一种动态的方式，在运行期间把功能动态地添加到对象结构中去。

有些朋友可能会想起装饰模式，装饰模式可以实现为一个对象透明地添加功能，但装饰模式基本上是在现有功能的基础之上进行功能添加，实际上是对现有功能的加强或者改造，并不是在现有功能不改动的情况下，为对象添加新的功能。

看来需要另外寻找新的解决方式了，可以应用访问者模式来解决这个问题。访问者模式实现的基本思路如下。

首先定义一个接口来代表要新加入的功能，为了通用，也就是定义一个通用的功能方法来代表新加入的功能。

在对象结构上添加一个方法，作为通用的功能方法，也就是可以代表被添加的功能，在这个方法中传入具体的实现新功能的对象。

在对象结构的具体实现对象中实现这个方法，回调传入具体的实现新功能的对象，就相当于调用到新功能上了。

接下来的步骤就是提供实现新功能的对象。

最后再提供一个能够循环访问整个对象结构的类，让这个类来提供符合客户端业务需求的方法，来满足客户端调用的需要。

这样一来，只要提供实现新功能的对象给对象结构，就可以为这些对象添加新的功能，由于在对象结构中定义的方法是通用的功能方法，所以什么新功能都可以加入。

25.2.2 访问者模式的结构和说明

访问者模式的结构如图 25.3 所示。

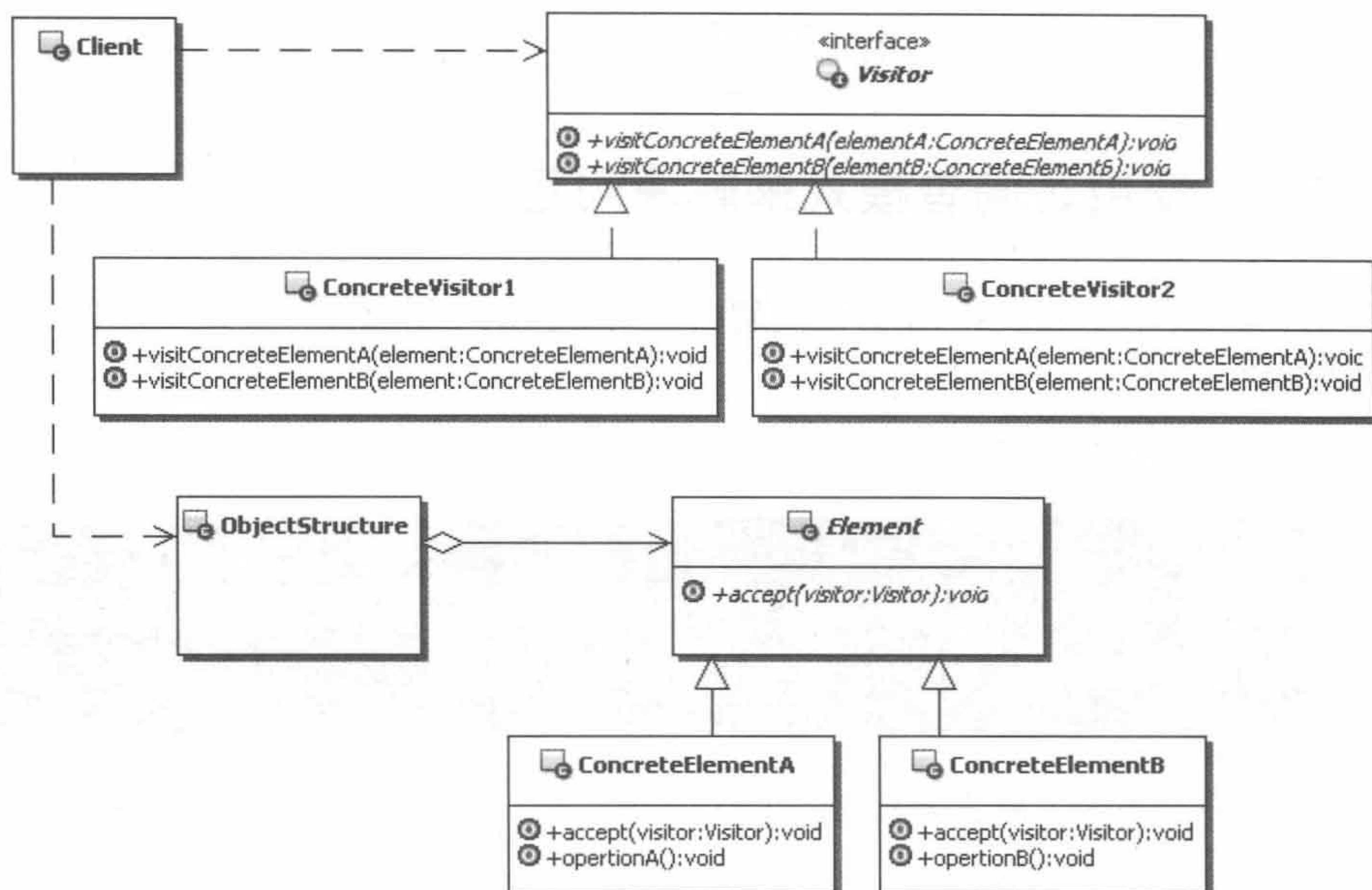


图 25.3　访问者模式结构示意图

- Visitor：访问者接口，为所有的访问者对象声明一个 visit 方法，用来代表为对象结构添加的功能，理论上可以代表任意的功能。
- ConcreteVisitor：具体的访问者实现对象，实现要真正被添加到对象结构中的功能。
- Element：抽象的元素对象，对象结构的顶层接口，定义接受访问的操作。
- ConcreteElement：具体元素对象，对象结构中具体的对象，也是被访问的对象，通常会回调访问者的真实功能，同时开放自身的数据供访问者使用。
- ObjectStructure：对象结构，通常包含多个被访问的对象，它可以遍历多个被访问的对象，也可以让访问者访问它的元素。可以是一个复合或是一个集合，如一个列表或无序集合。

注意　**但是请注意：这个 ObjectStructure 并不是我们在前面讲到的对象结构，前面一直讲的对象结构是指的一系列对象的定义结构，是概念上的东西；而 ObjectStructure 可以看成是对象结构中的一系列对象的一个集合，是用来辅助客户端访问这一系列对象的。为了不造成大家的困惑，所以后面提到 ObjectStructure 的时候，就用英文名称来代替，不把它翻译成中文。**

25.2.3　访问者模式示例代码

（1）首先需要定义一个接口来代表要新加入的功能，把它称作访问者，访问谁呢？当然是访问对象结构中的对象了。既然是访问，不能空手而去吧，这些访问者在进行访问的时候，就会携带新的功能。也就是说，访问者携带着需要添加的新的功能去访问对象结构中的对象，就相当于给对象结构中的对象添加了新的功能。示例代码如下：

```
/**
 * 访问者接口
 */
public interface Visitor {
    /**
     * 访问元素 A，相当于给元素 A 添加访问者的功能
     * @param elementA 元素 A 的对象
     */
    public void visitConcreteElementA(ConcreteElementA elementA);
    /**
     * 访问元素 B，相当于给元素 B 添加访问者的功能
     * @param elementB 元素 B 的对象
     */
    public void visitConcreteElementB(ConcreteElementB elementB);
}
```

（2）然后看看抽象的元素对象定义。示例代码如下：

```
/**
 * 被访问的元素的接口
 */
public abstract class Element {
    /**
     * 接受访问者的访问
     * @param visitor 访问者对象
     */
    public abstract void accept(Visitor visitor);
}
```

（3）再来看看元素对象的具体实现。

先看看元素 A 的实现。示例代码如下：

```
/**
 * 具体元素的实现对象
 */
public class ConcreteElementA extends Element {
    public void accept(Visitor visitor) {
        //回调访问者对象的相应方法
        visitor.visitConcreteElementA(this);
    }
    /**
     * 示例方法，表示元素已有的功能实现
     */
```

```
    public void opertionA(){
        //已有的功能实现
    }
}
```

再看看元素 B 的实现。示例代码如下：

```
/**
 * 具体元素的实现对象
 */
public class ConcreteElementB extends Element {
    public void accept(Visitor visitor) {
        //回调访问者对象的相应方法
        visitor.visitConcreteElementB(this);
    }
    /**
     * 示例方法，表示元素已有的功能实现
     */
    public void opertionB(){
        //已有的功能实现
    }
}
```

（4）接下来看看访问者的具体实现。

先看看访问者 1 的实现。示例代码如下：

```
/**
 * 具体的访问者实现
 */
public class ConcreteVisitor1 implements Visitor {
    public void visitConcreteElementA(ConcreteElementA element) {
        //把要访问 ConcreteElementA 时，需要执行的功能实现在这里
        //可能需要访问元素已有的功能，比如：
        element.opertionA();
    }
    public void visitConcreteElementB(ConcreteElementB element) {
        //把要访问 ConcreteElementB 时，需要执行的功能实现在这里
        //可能需要访问元素已有的功能，比如：
        element.opertionB();
    }
}
```

访问者 2 的实现和访问者 1 的示意代码是一样的，就不再赘述。

（5）该来看看 ObjectStructure 的实现了。示例代码如下：

```
/**
 * 对象结构，通常在这里对元素对象进行遍历，让访问者能访问到所有的元素
 */
public class ObjectStructure {
    /**
     * 示意，表示对象结构，可以是一个组合结构或是集合
     */
    private Collection<Element> col = new ArrayList<Element>();
    /**
     * 示意方法，提供给客户端操作的高层接口
     * @param visitor 客户端需要使用的访问者
     */
    public void handleRequest(Visitor visitor){
        //循环对象结构中的元素，接受访问
        for(Element ele : col){
            ele.accept(visitor);
        }
    }
    /**
     * 示意方法，组建对象结构，向对象结构中添加元素
     * 不同的对象结构有不同的构建方式
     * @param ele 加入到对象结构的元素
     */
    public void addElement(Element ele){
        this.col.add(ele);
    }
}
```

（6）最后来看看客户端的示意实现。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建 ObjectStructure
        ObjectStructure os = new ObjectStructure();
        //创建要加入对象结构的元素
        Element eleA = new ConcreteElementA();
        Element eleB = new ConcreteElementB();
        //把元素加入对象结构
        os.addElement(eleA);
        os.addElement(eleB);
        //创建访问者
        Visitor visitor = new ConcreteVisitor1();
```

```
            //调用业务处理的方法
            os.handleRequest(visitor);
        }
}
```

25.2.4 使用访问者模式重写示例

要使用访问者模式来重写示例，首先就要按照访问者模式的结构，分离出两个类层次来，一个是对应于元素的类层次，一个是对应于访问者的类层次。

对于对应于元素的类层次，现在已经有了，就是客户的对象层次。而对应于访问者的类层次，现在还没有，不过，按照访问者模式的结构，应该是先定义一个访问者接口，然后把每种业务实现成为一个单独的访问者对象，也就是说应该使用一个访问者对象来实现对客户的偏好分析，而用另外一个访问者对象来实现对客户的价值分析。

在分离好两个类层次以后，为了方便客户端的访问，定义一个 ObjectStructure，其实就类似于前面示例中客户管理的业务对象。新的示例的结构如图 25.4 所示。

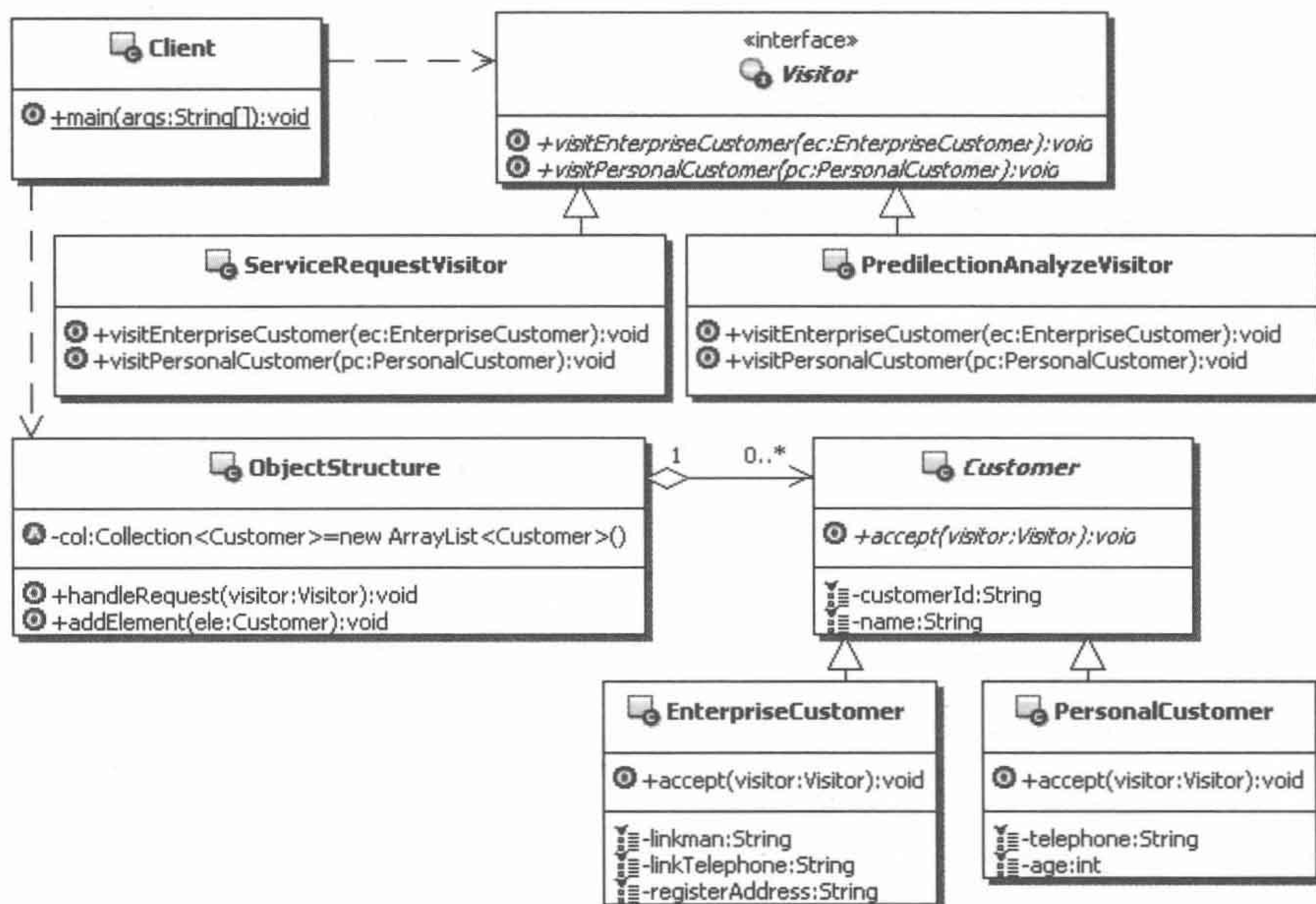


图 25.4　使用访问者模式的示例程序结构示意图

仔细查看图 25.4 所示的程序结构示意图，细心的朋友会发现，在图上没有出现对客户进行价值分析的功能了。这是为了示范“使用访问者模式来实现示例功能以后，可以很容易地给对象结构增加新的功能”，所以先不做这个功能，等都实现好了，再来扩展这个功能。接下来还是看看代码实现，以更好地体会访问者模式。

（1）先来看看 Customer 的代码，Customer 相当于访问者模式中的 Element，它的实

现和以前相比有以下改变。

- 新增一个接受访问者访问的方法。
- 把能够分离出去放到访问者中实现的方法，从 Customer 中删除，包括：客户提出服务请求的方法、对客户进行偏好分析的方法、对客户进行价值分析的方法等。

示例代码如下：

```
public abstract class Customer {
    private String customerId;
    private String name;
                                        属性对应的 getter/setter 方法，为
                                        了篇幅关系，省略了
     /**
      * 接受访问者的访问
      * @param visitor 访问者对象
      */
    public abstract void accept(Visitor visitor);
}
```

（2）下面来看看两种客户的实现。先来看看企业客户的实现。示例代码如下：

```
public class EnterpriseCustomer extends Customer{
    private String linkman;
    private String linkTelephone;
    private String registerAddress;
                                属性对应的 getter/setter 方法，为
                                了篇幅关系，省略了

    public void accept(Visitor visitor) {
        //回调访问者对象的相应方法
        visitor.visitEnterpriseCustomer(this);
    }
}
```

再来看看个人客户的实现。示例代码如下：

```
public class PersonalCustomer extends Customer{
    private String telephone;
    private int age;
                                属性对应的 getter/setter 方法，为
                                了篇幅关系，省略了
```

```
    public void accept(Visitor visitor) {
        //回调访问者对象的相应方法
        visitor.visitPersonalCustomer(this);
    }
}
```

（3）下面来看看访问者的接口定义。示例代码如下：

```
/**
 * 访问者接口
 */
public interface Visitor {
    /**
     * 访问企业客户，相当于给企业客户添加访问者的功能
     * @param ec 企业客户的对象
     */
    public void visitEnterpriseCustomer(EnterpriseCustomer ec);
    /**
     * 访问个人客户，相当于给个人客户添加访问者的功能
     * @param pc 个人客户的对象
     */
    public void visitPersonalCustomer(PersonalCustomer pc);
}
```

（4）下面来看看各个访问者的实现，每个访问者对象负责一类的功能处理。先来看看实现客户提出服务请求功能的访问者。示例代码如下：

```
/**
 * 具体的访问者，实现客户提出服务请求的功能
 */
public class ServiceRequestVisitor implements Visitor {
    public void visitEnterpriseCustomer(EnterpriseCustomer ec){
        //企业客户提出的具体服务请求
        System.out.println(ec.getName()+"企业提出服务请求");
    }
    public void visitPersonalCustomer(PersonalCustomer pc){
        //个人客户提出的具体服务请求
        System.out.println("客户"+pc.getName()+"提出服务请求");
    }
}
```

接下来看看实现对客户偏好分析功能的访问者。示例代码如下：

```
/**
```

```
 * 具体的访问者，实现对客户的偏好分析
 */
public class PredilectionAnalyzeVisitor implements Visitor {
    public void visitEnterpriseCustomer(EnterpriseCustomer ec){
        //根据以往购买的历史、潜在购买意向
        //以及客户所在行业的发展趋势、客户的发展预期等的分析
        System.out.println("现在对企业客户"+ec.getName()
                                            +"进行产品偏好分析");
    }
    public void visitPersonalCustomer(PersonalCustomer pc){
        System.out.println("现在对个人客户"+pc.getName()
                                            +"进行产品偏好分析");
    }
}
```

（5）下面来看看 ObjectStructure 的实现。示例代码如下：

```
public class ObjectStructure {
    /**
     * 要操作的客户集合
     */
    private Collection<Customer> col = new ArrayList<Customer>();
    /**
     * 提供给客户端操作的高层接口，具体的功能由客户端传入的访问者决定
     * @param visitor 客户端需要使用的访问者
     */
    public void handleRequest(Visitor visitor){
        //循环对象结构中的元素，接受访问
        for(Customer cm : col){
            cm.accept(visitor);
        }
    }
    /**
     * 组建对象结构，向对象结构中添加元素
     * 不同的对象结构有不同的构建方式
     * @param ele 加入到对象结构的元素
     */
    public void addElement(Customer ele){
        this.col.add(ele);
    }
}
```

（6）该来写个客户端测试一下了。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建 ObjectStructure
        ObjectStructure os = new ObjectStructure();
        //准备些测试数据，创建客户对象，并加入 ObjectStructure
        Customer cm1 = new EnterpriseCustomer();
        cm1.setName("ABC 集团");
        os.addElement(cm1);

        Customer cm2 = new EnterpriseCustomer();
        cm2.setName("CDE 公司");
        os.addElement(cm2);

        Customer cm3 = new PersonalCustomer();
        cm3.setName("张三");
        os.addElement(cm3);

        //客户提出服务请求，传入服务请求的 Visitor
        ServiceRequestVisitor srVisitor =
        new ServiceRequestVisitor();
        os.handleRequest(srVisitor);

        //要对客户进行偏好分析，传入偏好分析的 Visitor
        PredilectionAnalyzeVisitor paVisitor =
        new PredilectionAnalyzeVisitor();
        os.handleRequest(paVisitor);
    }
}
```

运行结果如下：

```
ABC 集团企业提出服务请求
CDE 公司企业提出服务请求
客户张三提出服务请求
现在对企业客户 ABC 集团进行产品偏好分析
现在对企业客户 CDE 公司进行产品偏好分析
现在对个人客户张三进行产品偏好分析
```

使用访问者模式重新实现了前面示例的功能，把各类相同的功能放在单独的访问者对象中，使得代码不再杂乱，系统结构也更清晰，能方便地维护了，解决了前面示例的一个问题。

还有一个问题，就是看看能不能方便地增加新的功能，前面在示例的时候，故意留下了一个对客户进行价值分析的功能没有实现，那么接下来就看看如何把这个功能增加到已有的系统中。在访问者模式中要给对象结构增加新的功能，只需要把新的功能实现成为访问者，然后在客户端调用的时候使用这个访问者对象来访问对象结构即可。

下面来看看实现对客户价值分析功能的访问者。示例代码如下：

```
/**
 * 具体的访问者，实现对客户价值分析
 */
public class WorthAnalyzeVisitor implements Visitor {
    public void visitEnterpriseCustomer(EnterpriseCustomer ec){
        //根据购买金额的大小、购买的产品和服务的多少、购买的频率等进行分析
        //企业客户的标准会比个人客户高
        System.out.println("现在对企业客户"+ec.getName()
                                    +"进行价值分析");
    }
    public void visitPersonalCustomer(PersonalCustomer pc){
        System.out.println("现在对个人客户"+pc.getName()
                                    +"进行价值分析");
    }
}
```

使用这个功能，只要在客户端添加以下代码即可。示例代码如下：

```
//要对客户进行价值分析，传入价值分析的 Visitor
WorthAnalyzeVisitor waVisitor = new WorthAnalyzeVisitor();
os.handleRequest(waVisitor);
```

测试看看，是否能正确地把这个功能加入到已有的程序结构中。

25.3 模 式 讲 解

25.3.1 认识访问者模式

1. 访问者的功能

访问者模式能给一系列对象透明地添加新功能，从而避免在维护期间对这一系列对象进行修改，而且还能变相实现复用访问者所具有的功能。

由于是针对一系列对象的操作，这也导致，如果只想给一系列对象中的部分对象添加功能，就会有些麻烦；而且要始终能保证把这一系列对象都调用到，不管是循环，还是递归，总之要让每个对象都要被访问到。

2. 调用通路

访问者之所以能实现"为一系列对象透明地添加新功能"，注意是透明的，也就是这一系列对象是不知道被添加功能的。

重要的就是依靠通用方法，访问者这边说要去访问，就提供一个访问的方法，如 visit 方法；而对象那边说，好的，我接受你的访问，提供一个接受访问的方法，如 accept 方法。这两个方法并不代表任何具体的功能，只是**构成一个调用的通路**，那么真正的功能实现在哪里呢？又如何调用到呢？

很简单，就在 accept 方法里面，回调 visit 的方法，从而回调到访问者的具体实现上，而这个访问者的具体实现的方法才是要添加的新的功能。

3. 两次分发技术

访问者模式能够实现在不改变对象结构的情况下，就可以给对象结构中的类增加功能，实现这个效果所使用的核心技术就是两次分发的技术。

在访问者模式中，当客户端调用 ObjectStructure 的时候，会遍历 ObjectStructure 中所有的元素，调用这些元素的 accept 方法，让这些元素来接受访问，这是请求的第一次分发；在具体的元素对象中实现 accept 方法的时候，会回调访问者的 visit 方法，等于请求被第二次分发了，请求被分发给访问者来进行处理，真正实现功能的正是访问者的 visit 方法。

两次分发技术具体的调用过程示意如图 25.5 所示。

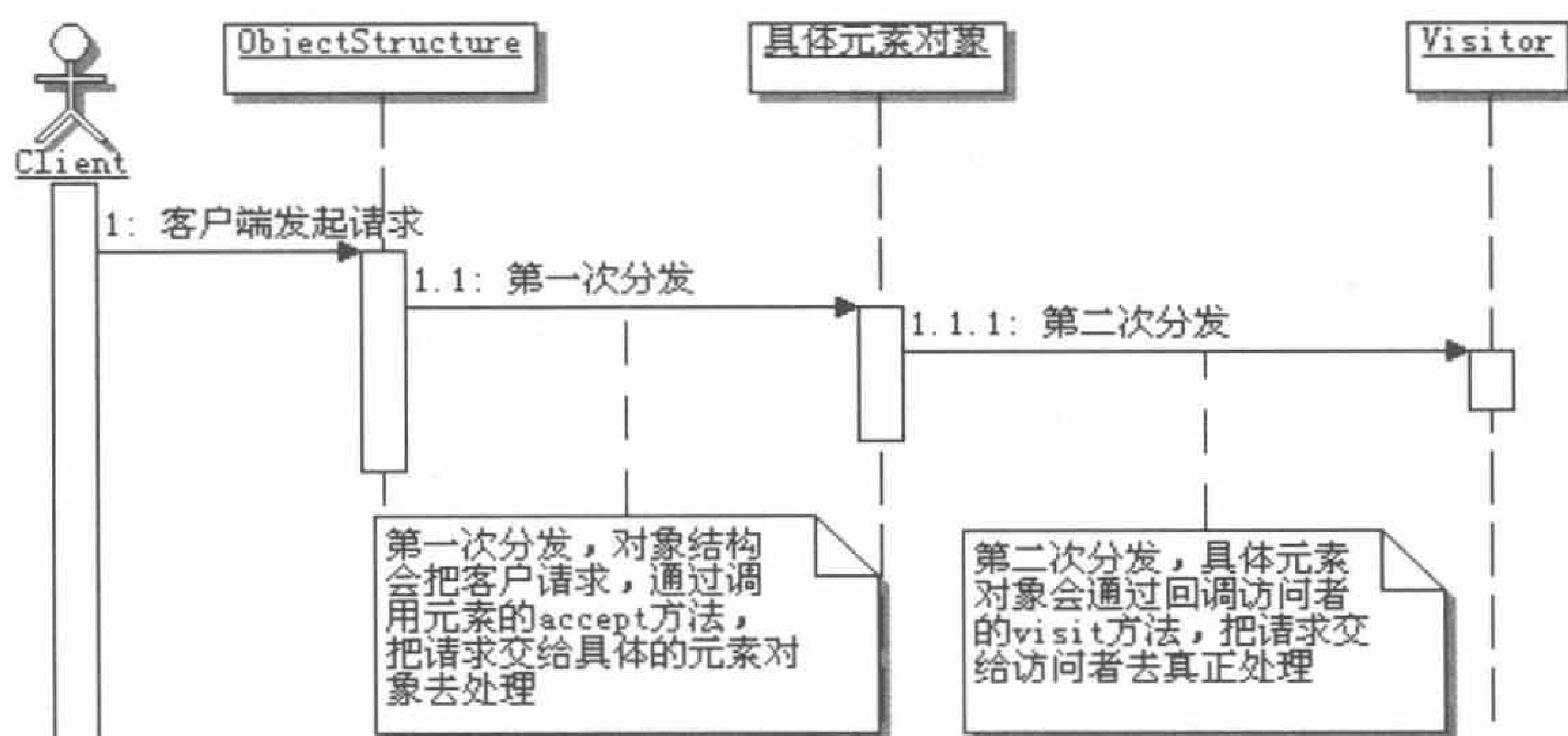


图 25.5 两次分发技术调用过程示意图

两次分发技术使得客户端的请求不再被静态地绑定在元素对象上，这个时候真正执行什么样的功能同时取决于访问者类型和元素类型，就算是同一种元素类型，只要访问者的类型不一样，最终执行的功能也会不一样，这样一来，就可以在元素对象不变的情况下，通过改变访问者的类型来改变真正执行的功能。

两次分发技术还有一个优点，就是可以在程序运行期间进行动态的功能组装和切换，只需在客户端调用时，组合使用不同的访问者对象实例即可。

从另一个层面思考，Java 回调技术也有点类似于两次分发技术。客户端调用某方法，这个方法就类似于 accept 方法，传入一个接口的实现对象，这个接口的实现对象就有点像是访问者，在方法内部，会回调这个接口的方法，就类似于调用访问者的 visit 方法，最终执行的还是接口的具体实现中实现的功能。

4. 为何不在 Component 中实现回调 visit 方法

在看上面示例的时候，细心的朋友会发现，在企业客户对象和个人客户对象中实现的 accept 方法从表面上看是相似的，都需要回调访问者的方法。可能就会有朋友想，

> **注意**
>
> **为什么不把回调访问者方法的调用语句放到父类中去，那样不就可以复用了吗？**
>
> **请注意，这是不可以的，虽然看起来是相似的语句，但其实是不同的，主要的玄机就在传入的 this 身上。this 是代表当前的对象实例的，在企业客户对象中传递的是企业客户对象的实例，在个人客户对象中传递的是个人客户对象的实例，这样在访问者的实现中，可以通过不同的对象实例来访问不同的实例对象的数据。**
>
> **如果把这句话放到父类中，那么传递的就是父类对象的实例，是没有子对象的数据的，因此这句话不能放到父类中去。**

5. 访问者模式的调用顺序示意图

访问者模式的调用顺序如图 25.6 所示。

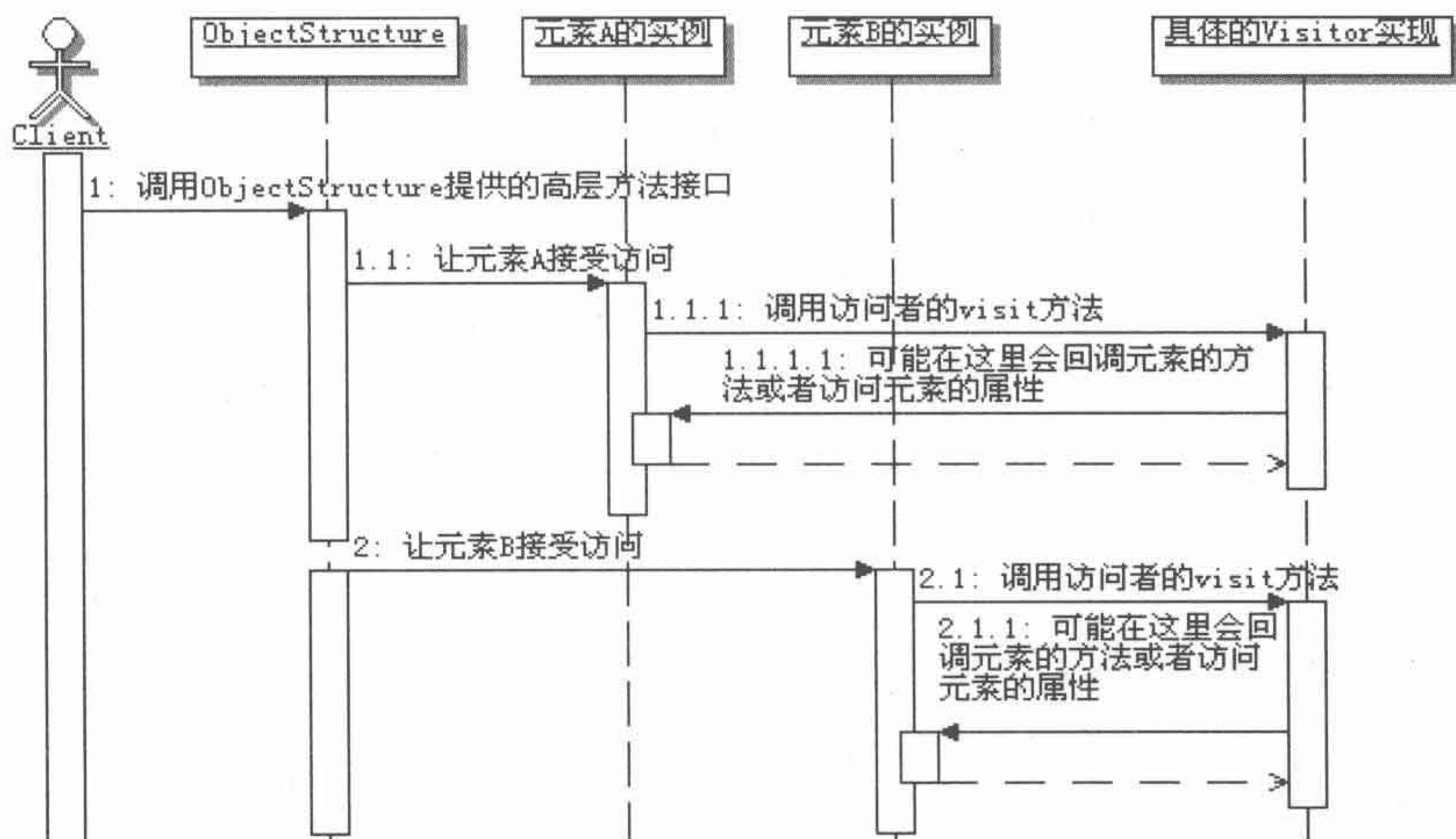


图 25.6　访问者模式调用顺序示意图

6. 空的访问方法

并不是所有的访问方法都需要实现，由于访问者模式默认的是访问对象结构中的所有元素，因此在实现某些功能的时候，如果不需要涉及到某些元素的访问方法，那么这些方法可以实现成为空的，比如，这个访问者只想处理组合对象，那么访问叶子对象的方法就可以为空，尽管还需要访问所有的元素对象。

还有一种就是有条件接受访问，在自己的 accept 方法中进行判断，满足要求的则接受，不满足要求的就相当于空的访问方法，什么都不用做。

25.3.2　操作组合对象结构

访问者模式一个很常见的应用，就是和组合模式结合使用，通过访问者模式来给由

组合模式构建的对象结构增加功能。

对于使用组合模式构建的组合对象结构，对外有一个统一的外观，要想添加新的功能也不是很困难，只要在组件的接口上定义新的功能就可以了，糟糕的是这样一来，需要修改所有的子类。而且，每次添加一个新功能，都需要修改组件接口，然后修改所有的子类。

为了让组合对象结构更灵活、更容易维护和有更好的扩展性，可以把它改造成访问者模式和组合模式组合来实现。这样在今后进行功能改造的时候，就不需要再改动这个组合对象结构了。

> 提示 **访问者模式和组合模式组合使用的思路：首先把组合对象结构中的功能方法分离出来，虽然维护组合对象结构的方法也可以分离出来，但是为了维持组合对象结构本身，这些方法还是放在组合对象结构中，然后把这些功能方法分别实现成访问者对象，通过访问者模式添加到组合的对象结构中去。**

下面通过访问者模式和组合模式组合来实现以下功能：输出对象的名称，在组合对象的名称前面添加“节点：”，在叶子对象的名称前面添加“叶子：”。

（1）先来定义访问者接口。

访问者接口非常简单，只需要定义访问对象结构中不同对象的方法。示例代码如下：

```
/**
 * 访问组合对象结构的访问者接口
 */
public interface Visitor {
    /**
     * 访问组合对象，相当于给组合对象添加访问者的功能
     * @param composite 组合对象
     */
    public void visitComposite(Composite composite);
    /**
     * 访问叶子对象，相当于给叶子对象添加访问者的功能
     * @param leaf 叶子对象
     */
    public void visitLeaf(Leaf leaf);
}
```

（2）改造组合对象的定义。

对已有的组合对象进行改造，添加通用的功能方法，当然在参数上需要传入访问者。先在组件定义上添加这个方法，然后到具体的实现类中去实现。除了新加这个方法外，组件定义没有其他改变。示例代码如下：

```
/**
```

```
 * 抽象的组件对象，相当于访问者模式中的元素对象
 */
public abstract class Component {
    /**
     * 接受访问者的访问
     * @param visitor 访问者对象
     */
    public abstract void accept(Visitor visitor);
    /**
     * 向组合对象中加入组件对象
     * @param child 被加入组合对象中的组件对象
     */
    public void addChild(Component child) {
        // 默认实现，抛出例外，叶子对象没这个功能，或子组件没有实现这个功能
        throw new UnsupportedOperationException(
                                "对象不支持这个功能");
    }
    /**
     * 从组合对象中移出某个组件对象
     * @param child 被移出的组件对象
     */
    public void removeChild(Component child) {
        // 默认实现，抛出例外，叶子对象没这个功能，或子组件没有实现这个功能
        throw new UnsupportedOperationException(
                                        "对象不支持这个功能");
    }
    /**
     * 返回某个索引对应的组件对象
     * @param index 需要获取的组件对象的索引，索引从 0 开始
     * @return 索引对应的组件对象
     */
    public Component getChildren(int index) {
        throw new UnsupportedOperationException(
                                        "对象不支持这个功能");
    }
}
```

添加的通用功能方法，可以代表任意被添加的功能，传入携带实际功能实现的访问者

（3）实现组合对象和叶子对象。

改变了组件定义，那么需要在组合类和叶子类上分别实现这个方法。

组合类中实现的时候，通常会循环让所有的子元素都接受访问，这样才能为所有的

对象都添加上新的功能。示例代码如下：

```
/**
 * 组合对象，可以包含其他组合对象或者叶子对象
 * 相当于访问者模式的具体 Element 实现对象
 */
public class Composite extends Component{
    public void ac
cept(Visitor visitor) {
        //回调访问者对象的相应方法
        visitor.visitComposite(this);
        //循环子元素，让子元素也接受访问
        for(Component c : childComponents){
            //调用子对象接受访问，变相实现递归
            c.accept(visitor);
        }
    }
    /**
     * 用来存储组合对象中包含的子组件对象
     */
    private List<Component> childComponents =
new ArrayList<Component>();
    /**
     * 组合对象的名字
     */
    private String name = "";
    /**
     * 构造方法，传入组合对象的名字
     * @param name 组合对象的名字
     */
    public Composite(String name){
        this.name = name;
    }
    public void addChild(Component child) {
        childComponents.add(child);
    }
    public String getName() {
        return name;
    }
}
```

接受访问者的访问，注意这里循环让所有的子元素都接受访问

叶子对象的基本实现。示例代码如下：

```
/**
 * 叶子对象，相当于访问者模式的具体 Element 实现对象
 */
public class Leaf extends Component{
    public void accept(Visitor visitor) {
        //回调访问者对象的相应方法
        visitor.visitLeaf(this);
    }
    /**
     * 叶子对象的名字
     */
    private String name = "";
    /**
     * 构造方法，传入叶子对象的名字
     * @param name 叶子对象的名字
     */
    public Leaf(String name){
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

（4）实现一个访问者。

组合对象结构已经改造好了，现在需要提供一个访问者的实现，它会实现真正的功能，也就是要添加到对象结构中的功能。示例代码如下：

```
/**
 * 具体的访问者，实现：输出对象的名称，在组合对象的名称前面添加"节点："
 * 在叶子对象的名称前面添加"叶子："
 */
public class PrintNameVisitor implements Visitor {
    public void visitComposite(Composite composite) {
        //访问到组合对象的数据
        System.out.println("节点："+composite.getName());
    }
    public void visitLeaf(Leaf leaf) {
        //访问到叶子对象的数据
        System.out.println("叶子："+leaf.getName());
    }
```

```
}
```

（5）访问所有元素对象的对象——ObjectStructure。

访问者是给一系列对象添加功能的，因此一个访问者需要访问所有的对象。为了方便遍历整个对象结构，通常会定义一个专门的类出来，在这个类中进行元素迭代访问，同时这个类提供客户端访问元素的接口。

对于这个示例，由于在组合对象结构中，已经实现了对象结构的遍历，本来是可以不需要 ObjectStructure 的，但是为了更清晰地展示访问者模式的结构，也为了今后的扩展或实现方便，还是定义一个 ObjectStructure。示例代码如下：

```
/**
 * 对象结构，通常在这里对元素对象进行遍历，让访问者能访问到所有的元素
 */
public class ObjectStructure {
    /**
     * 表示对象结构，可以是一个组合结构
     */
    private Component root = null;
    /**
     * 提供给客户端操作的高层接口
     * @param visitor 客户端需要使用的访问者
     */
    public void handleRequest(Visitor visitor){
        //让组合对象结构中的根元素，接受访问
        //在组合对象结构中已经实现了元素的遍历
        if(root!=null){
            root.accept(visitor);
        }
    }
    /**
     * 传入组合对象结构
     * @param ele 组合对象结构
     */
    public void setRoot(Component ele){
        this.root = ele;
    }
}
```

（6）写个客户端，来看看如何通过访问者去为对象结构添加新的功能。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //定义所有的组合对象
```

```
        Component root = new Composite("服装");
        Component c1 = new Composite("男装");
        Component c2 = new Composite("女装");
        //定义所有的叶子对象
        Component leaf1 = new Leaf("衬衣");
        Component leaf2 = new Leaf("夹克");
        Component leaf3 = new Leaf("裙子");
        Component leaf4 = new Leaf("套装");
        //按照树的结构来组合组合对象和叶子对象
        root.addChild(c1);
        root.addChild(c2);

        c1.addChild(leaf1);
        c1.addChild(leaf2);

        c2.addChild(leaf3);
        c2.addChild(leaf4);

        //创建 ObjectStructure
        ObjectStructure os = new ObjectStructure();
        os.setRoot(root);

        //调用 ObjectStructure 来处理请求功能
        Visitor psVisitor = new PrintNameVisitor();
        os.handleRequest(psVisitor);
    }
}
```

输出的效果如下：

```
节点：服装
节点：男装
叶子：衬衣
叶子：夹克
节点：女装
叶子：裙子
叶子：套装
```

看看结果，是不是期望的那样呢？

提示	**好好体会一下，想想访问者模式是如何实现动态地给组件添加功能的？尤其要想想，实现的机制是什么？真正实现新功能的地方在哪里？**

（7）现在的程序结构。

前面是分步的示范，大家已经体会了一番，接下来小结一下。

如同前面的示例，访问者的方法就相当于作用于组合对象结构中各个元素的操作，是一种通用的表达，同样的访问者接口和同样的方法，只要提供不同的访问者具体实现，就表示不同的功能。

同时在组合对象中，接受访问的方法，也是一个通用的表达，不管你是什么样的功能，统统接受就好了，然后回调回去执行真正的功能。这样一来，各元素的类就不用再修改了，只要提供不同的访问者实现，然后通过这个通用表达，就结合到组合对象中来了，相当于给所有的对象提供了新的功能。

示例的整体结构。如图 25.7 所示。

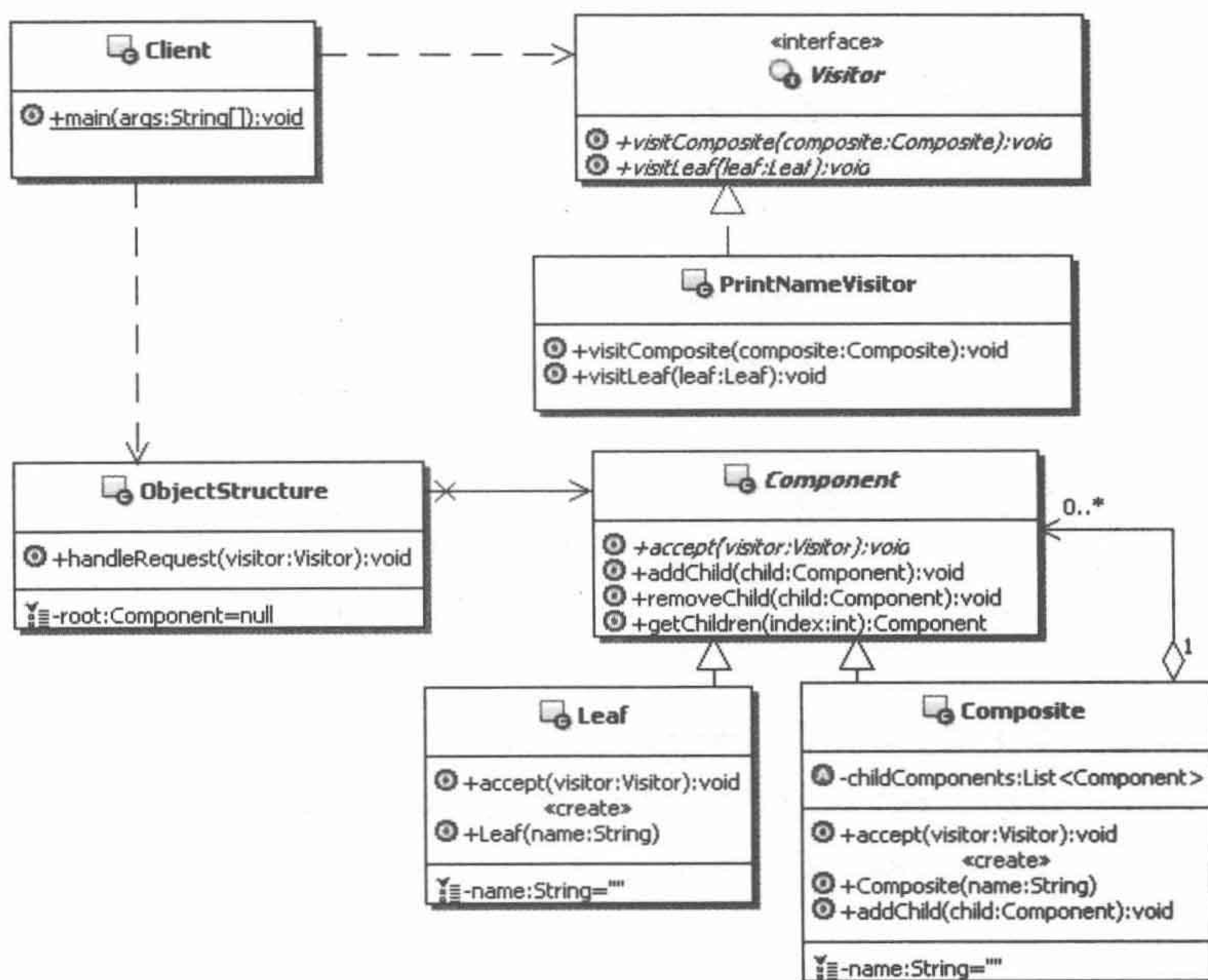


图 25.7　访问者模式结合组合模式的示例的结构示意图

25.3.3　谁负责遍历所有元素对象

在访问者模式中，访问者必须要能够访问到对象结构中的每个对象，因为访问者要为每个对象添加功能，为此特别在模式中定义一个 ObjectStructure，然后由 ObjectStructure 负责遍历访问一系列对象中的每个对象。

（1） 在 ObjectStructure 迭代所有的元素时，又分成以下两种情况。

- 元素的对象结构是通过集合来组织的，因此直接在 ObjectStructure 中对集合进行迭代，然后对每一个元素调用 accept 就可以了。如同 25.2.4 节的示例所采用的方式。
- 元素的对象结构是通过组合模式来组织的，通常可以构成对象树，这种情况一般就不需要在 ObjectStructure 中迭代了。而通常的做法是在组合对象的 accept 方法中，递归遍历它的子元素，然后调用子元素的 accept 方法，如同 25.3.2 节的示例中

Composite 的实现，在 accept 方法中进行递归调用子对象的操作。

（2）不需要 ObjectStructure 的时候。

在实际开发中，有一种典型的情况可以不需要 ObjectStructure 对象，那就是只有一个被访问对象的时候。只有一个被访问对象，当然就不需要使用 ObjectStructure 来组合和迭代了，只需调用这个对象就可以了。

事实上还有一种情况也可以不使用 ObjectStructure，比如上面访问的组合对象结构。从客户端的角度看，他访问的其实就是一个对象，因此可以把 ObjectStructure 去掉，然后直接从客户端调用元素的 accept 方法。

还是通过示例来说明。先把 ObjectStructure 类去掉。由于没有了 ObjectStructure，那么客户端调用的时候就直接调用组合对象结构根元素的 accept 方法。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //定义组件数据，组装对象树，跟刚才的测试一样，这里就省略了

        Visitor psVisitor = new PrintNameVisitor();
        root.accept(psVisitor);
    }
}
```

（3）有些时候，遍历元素的方法也可以放到访问者中，当然也是需要递归遍历它的子元素的。出现这种情况的主要原因是，想在访问者中实现特别复杂的遍历，访问者的实现依赖于对象结构的操作结果。

比如 25.3.2 节的示例，使用访问者模式和组合模式组合来实现了输出名称的功能，如果现在要实现把组合的对象结构按照树的形式输出，就要按照在组合模式中示例的那样，输出如下的树形结构：

```
+服装
 +男装
  -衬衣
  -夹克
 +女装
  -裙子
  -套装
```

要实现这个功能，在组合对象结构中遍历子对象的方式就比较难于实现了，因为要输出这个树形结构，需要控制每个对象在输出的时候，向后的退格数量，这个需要在对象结构的循环中来控制，这种功能可以选择在访问者当中去遍历对象结构。

来改造上面的示例，看看通过访问者来遍历元素如何实现这样的功能。

首先在 Composite 的 accept 实现中去除递归调用子对象的代码，同时添加一个让访问者访问到其所包含的子对象的方法。示例代码如下：

```
public class Composite extends Component{
```

```java
    //其他相同部分就省略了，只看变化的方法
    public void accept(Visitor visitor) {
        //回调访问者对象的相应方法
        visitor.visitComposite(this);
        for(Component c : childComponents){
            //调用子对象接受访问，变相实现递归
            c.accept(visitor);
        }
    }
public List<Component> getChildComponents() {
        return childComponents;
    }
}
```


然后新实现一个访问者对象，在相应的 visit 实现中，添加递归迭代所有子对象。示例代码如下：

```java
/**
 * 具体的访问者，实现：输出组合对象自身的结构
*/
public class PrintStructVisitor implements Visitor {
    /**
     * 用来累计记录对象需要向后退的格
     */
    private String preStr = "";
    public void visitComposite(Composite composite) {
        //先把自己输出去
        System.out.println(preStr+"+"+composite.getName());
        //如果还包含有子组件，那么就输出这些子组件对象
        if(composite.getChildComponents()!=null){
            //然后添加一个空格，表示向后缩进一个空格
            preStr+=" ";
            //输出当前对象的子对象了
            for(Component c : composite.getChildComponents()){
                //递归输出每个子对象
                c.accept(this);
            }
            //把循环子对象多加入的一个退格给去掉
            preStr = preStr.substring(0,preStr.length()-1);
        }
    }
    public void visitLeaf(Leaf leaf) {
```

```
        //访问到叶子对象的数据
        System.out.println(preStr+"-"+leaf.getName());
    }
}
```

写个客户端来测试一下看看，是否能实现要求的功能。示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //定义所有的组合对象过程跟上一个 client 是一样的，这里省略了
        //以调用根元素的方法来接受请求功能
        Visitor psVisitor = new PrintStructVisitor();
        root.accept(psVisitor);
    }
}
```

25.3.4　访问者模式的优缺点

访问者模式有以下优点。

- 好的扩展性

  能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。

- 好的复用性

  可以通过访问者来定义整个对象结构通用的功能，从而提高复用程度。

- 分离无关行为

  可以通过访问者来分离无关的行为，把相关的行为封装在一起，构成一个访问者，这样每一个访问者的功能都比较单一。

访问者模式有以下缺点。

- 对象结构变化很困难

  不适用于对象结构中的类经常变化的情况，因为对象结构发生了改变，访问者的接口和访问者的实现都要发生相应的改变，代价太高。

- 破坏封装

  访问者模式通常需要对象结构开放内部数据给访问者和 ObjectStructrue，这破坏了对象的封装性。

25.3.5　思考访问者模式

1. 访问者模式的本质

> 访问者模式的本质：**预留通路，回调实现。**

仔细思考访问者模式，它的实现主要是通过预先定义好调用的通路，在被访问的对象上定义 accept 方法，在访问者的对象上定义 visit 方法；然后在调用真正发生的时候，通过两次分发技术，利用预先定义好的通路，回调到访问者具体的实现上。

明白了访问者模式的本质，就可以在定义一些通用功能，或者设计工具类的时候让访问者模式派上大用场。你可以把已经实现好的一些功能作为已有的对象结构，因为在今后可能会根据实际需要为它们增加新的功能，甚至希望开放接口来让其他开发人员扩展这些功能，所以你可以用访问者模式来设计，在这个对象结构上预留好通用的调用通路，在以后添加功能，或者是其他开发人员来扩展的时候，只需要提供新的访问者实现，就能够很好地加入到系统中来了。

2. 何时选用访问者模式

建议在以下情况中选用访问者模式。

- 如果想对一个对象结构实施一些依赖于对象结构中具体类的操作，可以使用访问者模式。
- 如果想对一个对象结构中的各个元素进行很多不同的而且不相关的操作，为了避免这些操作使类变得杂乱，可以使用访问者模式。把这些操作分散到不同的访问者对象中去，每个访问者对象实现同一类功能。
- 如果对象结构很少变动，但是需要经常给对象结构中的元素对象定义新的操作，可以使用访问者模式。

25.3.6 相关模式

- 访问者模式和组合模式

  这两个模式可以组合使用。

  如同前面示例的那样，通过访问者模式给组合对象预留下扩展功能的接口，使得为组合模式的对象结构添加功能非常容易。
- 访问者模式和装饰模式

  这两个模式从表面上看功能有些相似，都能够实现在不修改原对象结构的情况下修改原对象的功能。但是装饰模式更多的是实现对已有功能的加强、修改或者完全全新实现；而访问者模式更多的是实现为对象结构添加新的功能。
- 访问者模式和解释器模式

  这两个模式可以组合使用。

  解释器模式在构建抽象语法树的时候，是使用组合模式来构建的，也就是说解释器模式解释并执行的抽象语法树是一个组合对象结构，这个组合对象结构是很少变动的，但是可能经常需要为解释器增加新的功能，实现对同一对象结构的不同解释和执行的功能，这正是访问者模式的优势所在，因此在使用解释器模式的时候通常会组合访问者模式来使用。

A.1 设计模式和设计原则

A.1.1 设计模式和设计原则的关系

面向对象的分析设计有很多原则，这些原则大多从思想层面给我们指出了面向对象分析设计的正确方向，是我们进行面向对象分析设计时应该尽力遵守的准则。

提示 **而设计模式已经是针对某个场景下某些问题的某个解决方案。也就是说这些设计原则是思想上的指导，而设计模式是实现上的手段，因此设计模式也应该遵守这些原则，换句话说，设计模式就是这些设计原则的一些具体体现。**

A.1.2 为何不重点讲解设计原则

既然设计模式是这些设计原则的具体体现，那也就意味着设计模式的思想上的根就是这些设计原则了，没错，可以这么认为。

这样一来，有些朋友就会很疑惑了，那么为何不重点讲讲设计原则呢？对于这个问题，我们有如下的考虑。

- 设计原则本身是从思想层面上进行指导，本身是高度概括和原则性的。只是一个设计上的大体方向，其具体实现并非只有设计模式这一种。理论上来说，可以在相同的原则指导下，做出很多不同的实现来。
- 每一种设计模式并不是单一地体现某一个设计原则。事实上，很多设计模式都是融合了很多个设计原则的思想，并不好特别强调设计模式对某个或者是某些设计原则的体现。而且每个设计模式在应用的时候也会有很多的考量，不同使用场景下，突出体现的设计原则也可能是不一样的。
- 这些设计原则只是一个建议指导。事实上，在实际开发中，很少做到完全遵守，总是在有意无意地违反一些或者是部分设计原则。设计工作本来就是一个不断权衡的工作，有句话说得很好：**"设计是一种危险的平衡艺术"**。设计原则只是一个指导，有些时候，还要综合考虑业务功能、实现的难度、系统性能、时间与空间等很多方面的问题。
- 设计模式本身就已经很复杂了。在一本书中很难再去深入地探讨这些设计原则，这样也避免出现过多的重点内容，导致大家无所适从。

本书的目标是想与朋友们深入地探讨设计模式而不是设计原则。因此我们选择不去深入讲解设计原则。事实上，即使你不懂这些设计原则，对本书的阅读也没有太大的影响，只是在一些问题认识的深度上可能会有一点阻碍。

基于同样的道理，这里也没有过多从重构的角度去讲述设计模式。

当然，在某些设计模式中，明显地体现了某些设计原则，我们也还是会与朋友们一起来讨论和分享的。

> 提示 **这里为不熟悉这些设计原则的朋友，简要准备了一些常见的、基本的面向对象设计原则的知识，可以先阅读这些内容，然后再回去看设计模式的内容，可能会有一定的帮助。但请注意，这并不是面向对象设计原则的全部，更多的知识，有机会再与大家一起分享。**

A.2 常见的面向对象设计原则

A.2.1 单一职责原则 SRP（Single Responsibility Principle）

所谓单一职责原则，指的是，一个类应该仅有一个引起它变化的原因。

这里变化的原因就是所说的“职责”，如果一个类有多个引起它变化的原因，那么也就意味着这个类有多个职责，再进一步说，就是把多个职责耦合在一起了。

这会造成职责的相互影响，可能一个职责的变化，会影响到其他职责的实现，甚至引起其他职责随着变化，这种设计是很脆弱的。

这个原则看起来是最简单和最好理解的，但是实际上是很难完全做到的，难点在于如何区分 “职责”。这是个没有标准量化的东西，哪些算职责、到底这个职责有多大的粒度、这个职责如何细化等。因此，在实际开发中，这个原则也是最容易违反的。

A.2.2 开放-关闭原则 OCP（Open-Closed Principle）

所谓开放—关闭原则，指的是，一个类应该对扩展开放，对修改关闭。一般也被简称为开闭原则，开闭原则是设计中非常核心的一个原则。

开闭原则要求的是，类的行为是可以扩展的，而且是在不修改已有代码的情况下进行扩展，也不必改动已有的源代码或者二进制代码。

看起来好像是矛盾的，怎么样才能实现呢？

实现开闭原则的关键就在于合理地抽象、分离出变化与不变化的部分，为变化的部分预留下可扩展的方式，比如，钩子方法或是动态组合对象等。

这个原则看起来也很简单。但事实上，一个系统要全部做到遵守开闭原则，几乎是不可能的，也没这个必要。适度的抽象可以提高系统的灵活性，使其可扩展、可维护，但是过度地抽象，会大大增加系统的复杂程度。应该在需要改变的地方应用开闭原则就可以了，而不用到处使用，从而陷入过度设计。

A.2.3 里氏替换原则 LSP（Liskov Substitution Principle）

所谓里氏替换原则，指的是，子类型必须能够替换掉它们的父类型。这很明显是一种多态的使用情况，它可以避免在多态的应用中，出现某些隐蔽的错误。

事实上，当一个类继承了另外一个类，那么子类就拥有了父类中可以继承下来的属性和操作。理论上来说，此时使用子类型去替换掉父类型，应该不会引起原来使用父类型的程序出现错误。

但是，很不幸的是，在某些情况下是会出现问题的。比如，如果子类型覆盖了父类型的某些方法，或者是子类型修改了父类型某些属性的值，那么原来使用父类型的程序就可能会出现错误，因为在运行期间，从表面上看，它调用的是父类型的方法，需要的是父类型方法实现的功能，但是实际运行调用的却是子类型覆盖实现的方法，而该方法和父类型的方法并不一样，于是导致错误的产生。

从另外一个角度来说，里氏替换原则是实现开闭的主要原则之一。开闭原则要求对扩展开放，扩展的一个实现手段就是使用继承；而里氏替换原则是保证子类型能够正确替换父类型，只有能正确替换，才能实现扩展，否则扩展了也会出现错误。

A.2.4 依赖倒置原则 DIP（Dependence Inversion Principle）

所谓依赖倒置原则，指的是，要依赖于抽象，不要依赖于具体类。要做到依赖倒置，典型的应该做到：

- 高层模块不应该依赖于底层模块，二者都应该依赖于抽象。
- 抽象不应该依赖于具体实现，具体实现应该依赖于抽象。

很多人觉得，层次化调用的时候，应该是高层调用“底层所拥有的接口”，这是一种典型的误解。事实上，一般高层模块包含对业务功能的处理和业务策略选择，应该被重用，是高层模块去影响底层的具体实现。

因此，这个底层的接口应该是由高层提出的，然后由底层实现的。也就是说底层的接口的所有权在高层模块，因此是一种所有权的倒置。

倒置接口所有权，这就是著名的 Hollywood（好莱坞）原则：不要找我们，我们会联系你。

A.2.5 接口隔离原则 ISP（Interface Segregation Principle）

所谓接口隔离原则，指的是，不应该强迫客户依赖于他们不用的方法。

这个原则用来处理那些比较“庞大”的接口，这种接口通常会有较多的操作声明，涉及到很多的职责。客户在使用这样的接口的时候，通常会有很多他不需要的方法，这些方法对于客户来讲，就是一种接口污染，相当于强迫用户在一大堆“垃圾方法”中去寻找他需要的方法。

因此，这样的接口应该被分离，应该按照不同的客户需要来分离成为针对客户的接口。这样的接口中，只包含客户需要的操作声明，这样既方便了客户的使用，也可以避免因误用接口而导致的错误。

分离接口的方式，除了直接进行代码分离之外，还可以使用委托来分离接口，在能够支持多重继承的语言中，还可以采用多重继承的方式进行分离。

A. 2. 6 最少知识原则 LKP（Least Knowledge Principle）

所谓最少知识原则，指的是，只和你的朋友谈话。

这个原则用来指导我们在设计系统的时候，应该尽量减少对象之间的交互，对象只和自己的朋友谈话，也就是只和自己的朋友交互，从而松散类之间的耦合。通过松散类之间的耦合来降低类之间的相互依赖，这样在修改系统的某一个部分的时候，就不会影响其他的部分，从而使得系统具有更好的可维护性。

那么究竟哪些对象才能被当作朋友呢？最少知识原则提供了一些指导。

- 当前对象本身。
- 通过方法的参数传递进来的对象。
- 当前对象所创建的对象。
- 当前对象的实例变量所引用的对象。
- 方法内所创建或实例化的对象。

总之，最少知识原则要求我们的方法调用必须保持在一定的界限范围之内，尽量减少对象的依赖关系。

A. 2. 7 其他原则

除了上面提到的这些原则，还有一些大家都熟知的原则，比如：

- 面向接口编程；
- 优先使用组合，而非继承。

当然也还有很多大家不是很熟悉的原则，比如：

- 一个类需要的数据应该隐藏在类的内部；
- 类之间应该零耦合，或者只有传导耦合，换句话说，类之间要么没有关系，要么只使用另一个类的接口提供的操作；
- 在水平方向上尽可能统一地分布系统功能；

还有很多，这里就不去详细讨论这些内容了。

读书笔记

B.1 UML 基础

由于本书用 UML 来表达模式的结构和基本的运行顺序示意，特此为不熟悉 UML 的朋友准备了一些相关的 UML 快速入门知识。

这里只是 UML 知识的一小部分，如果需要了解更多的 UML 知识，请参阅 UML 的学习文档，UML 的网站 http://www.uml.org/是个好去处。

B.1.1 UML 是什么

UML 是一种标准的图形化建模语言，它是面向对象分析与设计的一种标准表示。

（1）UML 是一种**语言。**

从上面的定义可以看出，就其本质，UML 是一种语言，既然是语言，那就是用来交流的，UML 用来在哪些人员之间进行交流呢？很明显，UML 主要是在软件开发的整个生命周期所涉及到的人员之间进行交流的语言。

（2）UML 是一种**建模**语言。

那么什么是建模呢？

模型是用某种工具对事物的一种表达方式，通常会表达出事物最重要的方面而简化或忽略其他方面。比如常见的工程模型、飞机模型、车辆模型等。

模型在软件上主要的作用是，可以在一定的抽象层次上，使人们通过对模型的分析和研究，来制定出最终的软件结构和内部的相互关系。

（3）UML 是一种**图形化**建模语言。

为什么要图形化呢？

很简单，图形化的东西直观、简单、准确，更有利于软件开发的整个生命周期所涉及到的人员之间进行交流。因为对于一个大型的软件项目，参与的人员很多，根本不可能相互用语言来交流，图形化是一个很好的方案。

（4）UML 是一种**标准**的图形化建模语言。

只有标准的东西，才会有更多的人学习和使用它，大家对同一表达的理解才会一样，才能真正达到相互交流的目的。

否则要是没有标准，大家各自为政，可能会出现同一个图形，大家有不同的认识和理解，那就没法交流了。

B.1.2 UML 历史

UML 出现在 1995 年，到 1997 年的时候，由 OMG 进行统一，并于同年由 OMG 全体成员通过采纳为标准，也就是 UML 1.1 版。

1998 年，对 UML 1.1 版进行了少量修改，推出了 UML 1.2 版，随后几年，陆续地推出了 1.3、1.4、1.5 等版本。

直到 2003 年 UML 2.0 被 OMG 接纳为标准，最新的 UML 2.0 的可用版本于 2005 年 7 月发布。

B.1.3 UML 能干什么

UML 主要用于对软件进行描述、可视化处理、构造和建立软件系统的文档，以方便对系统的理解、设计、浏览、配置、维护和信息控制。通过它，参与软件各个生命周期的人员可以很方便地交流。

B.1.4 UML 有什么

简单的说，UML 由视图构成，视图由图构成，图由图片组成，图片是模型元素的符号化。图是一个具体视图的组成部分，一种视图通常会包含多种图。

- 视图：描述完整系统中的一个抽象，用来显示这个系统中的一个特定的方面。
- 图：用来表示系统的一个特殊部分或某个方面。
- 模型元素：所有可以在图中使用的概念统称为模型元素。

在 UML 2.0 里面，视图被分成三个视图域：结构、动态和模型管理，具体的视图和图如表 B.1 所示。

表 B.1 UML 视图和图列表

主 要 的 域	视 图	图
结构	静态视图	类图
		对象图
	用例视图	用例图
	实现视图	组件图
	部署视图	部署图
动态	状态视图	状态图
	活动视图	活动图
	交互视图	顺序图
		协作图
模型管理	模型管理视图	类图

UML 的视图和图基本上都具有可扩展性，这些扩展能力有限但是很有用，包括约束、构造型和标记值，这里就不去介绍了。

本书主要用到了类图和顺序图，下面就简要地介绍一些关于类图和顺序图的基本知识，其他的图这里就不去涉及了，如有需要，请参阅相关资料。

B.2 类 图

B.2.1 类图的概念

类图是静态视图的图形表达方式，表示声明的静态模型元素，如类、类型和其内容，以及它们的相互关系。也就是说，类图是用来描述类以及类与类之间关系的一种 UML 图。

B.2.2 类图的基本表达

类图的基本模型元素如图 B.1 所示。

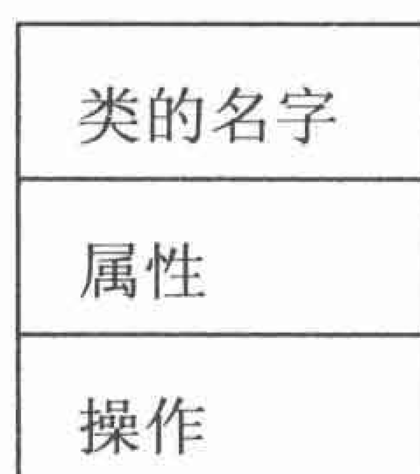


图 B.1 类图的基本图示

也就是说，一个类的图形表示为长方形，长方形又分成三个部分，分别是类名、属性定义和操作也就是方法定义。

1. 类名的定义

没有特殊要求，任何合法的名称都可以。类名通常为一个名词。为类命名时最好能够反映类所代表的问题的域中的概念，另外类的名字含义要准确、清楚。

2. 属性定义的基本语法

属性用来描述类所具有的特征。描述属性的语法格式为：

可见性属性名：类型名=初值

其中属性名和类型名是一定要有的，其他部分可选。

对于可见性：+表示 public，—表示 private，#表示 protected，没有符号就表示是默认的可见性。基本的示例如图 B.2 所示。

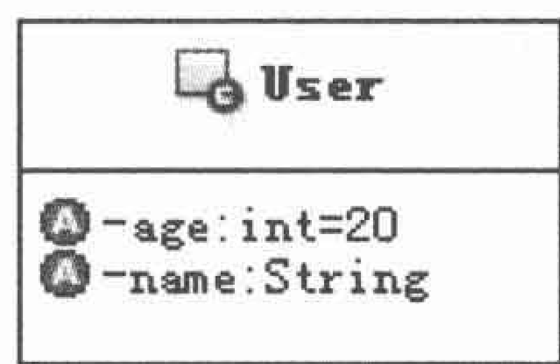


图 B.2 带属性的类图

> **注意**
> 注意一点：Attribute 和 Property 虽然都是表示类的属性，但是一些属性只是在类内部使用，不对外的，一般称这些属性为 Attribute，也有一些属性虽然是 private 的，但是会提供相应的 getter/setter 方法让外部来操作，把这些属性称为 Property。

具体的还是看个示例，给上面的类图添加一个名称为绰号，也就是"nickname"的 Property，如图 B.3 所示。

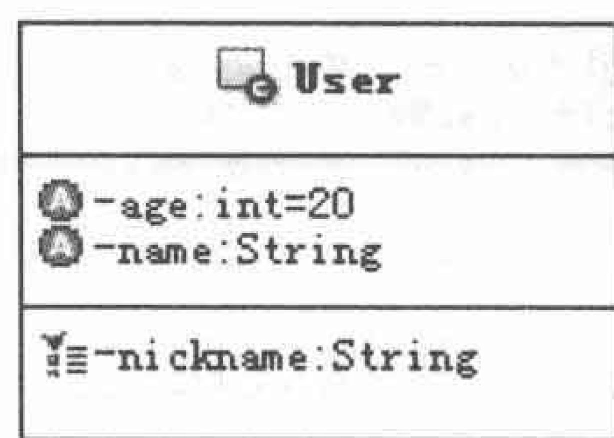


图 B.3　同时有 Attribute 和 Property 的类图

3. 操作定义的基本语法

操作用来描述类能干些什么事情，也就是我们通常说的方法。描述操作的语法格式为：

可见性 操作名（参数列表）：返回值类型

可见性和属性的描述方式一样，都是+表示 public，-表示 private，#表示 protected，没有符号就表示是默认的可见性。参数列表由多个参数构成，用逗号分隔，描述参数的语法格式为：

参数名 ：参数类型名

在图 B.3 所示的类图中添加一个方法：人能够按照指定的速度和持续时间进行跑步运动，如图 B.4 所示。

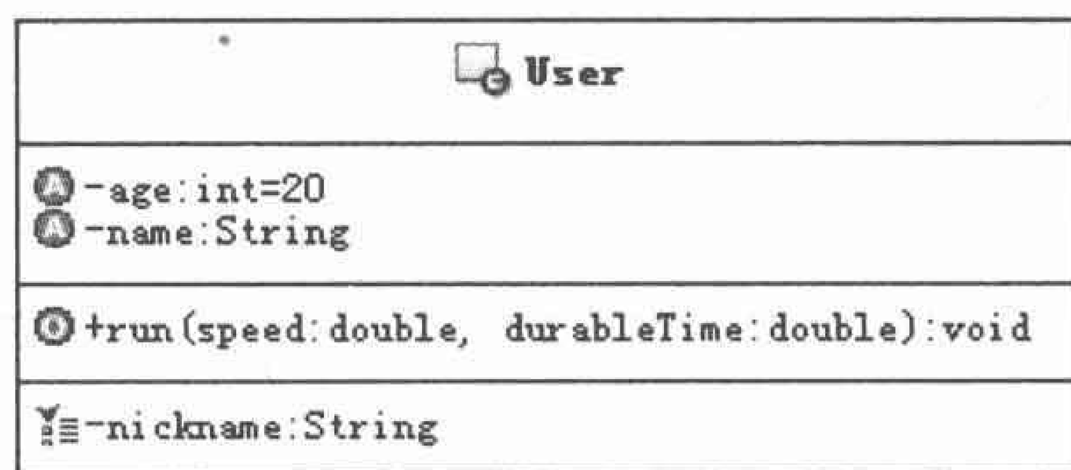


图 B.4　带操作的类图

4. Java 中 static 的表示

在类图中，如果属性或者方法是 static 的，那么在属性或者方法定义的下面，添加一条下划线表示是 static 的，如图 B.5 所示。

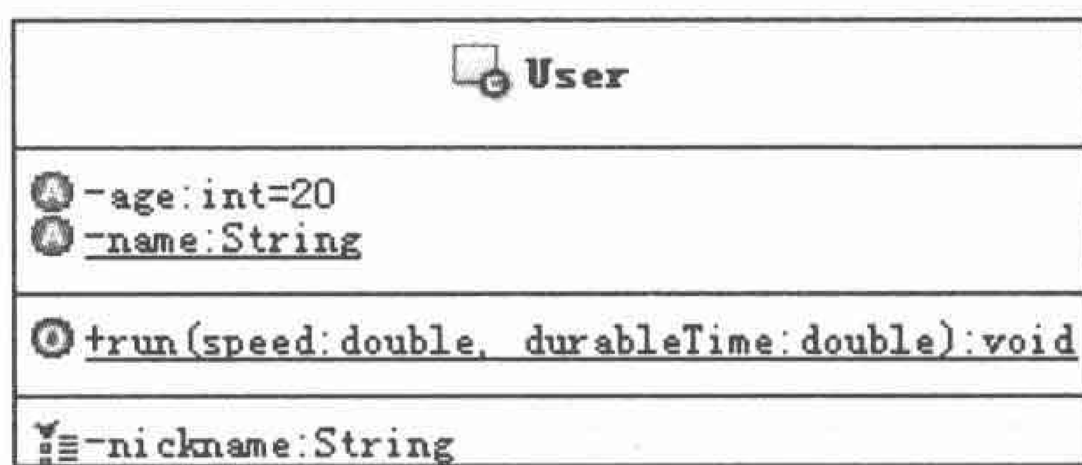


图 B.5　带 static 的类图

B.2.3　抽象类和接口

抽象类的表示是类名倾斜，抽象操作的表示是整条操作定义都倾斜，如图 B.6 所示。

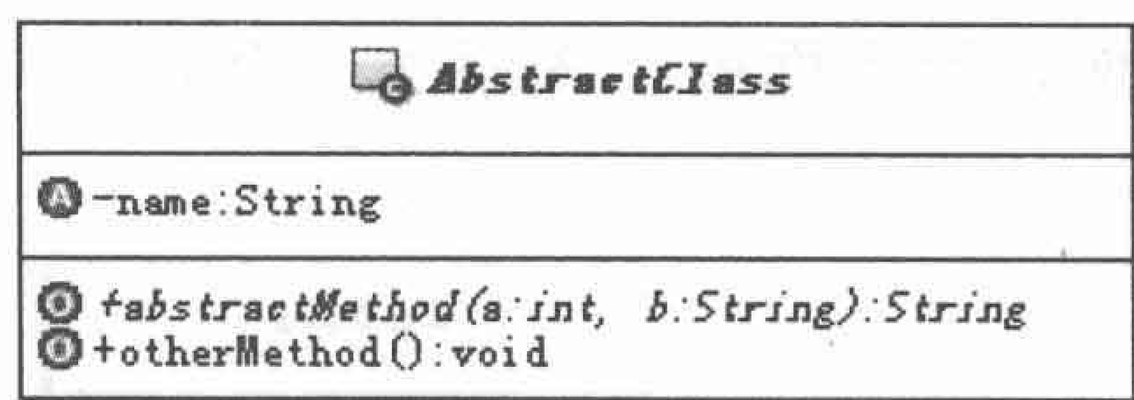


图 B.6　抽象类的类图

接口是一种特殊的抽象类，归根结底还是类，所以接口的表达基本语法和类是一样的，比如一个有创建用户和删除用户的接口，定义示例如图 B.7 所示。

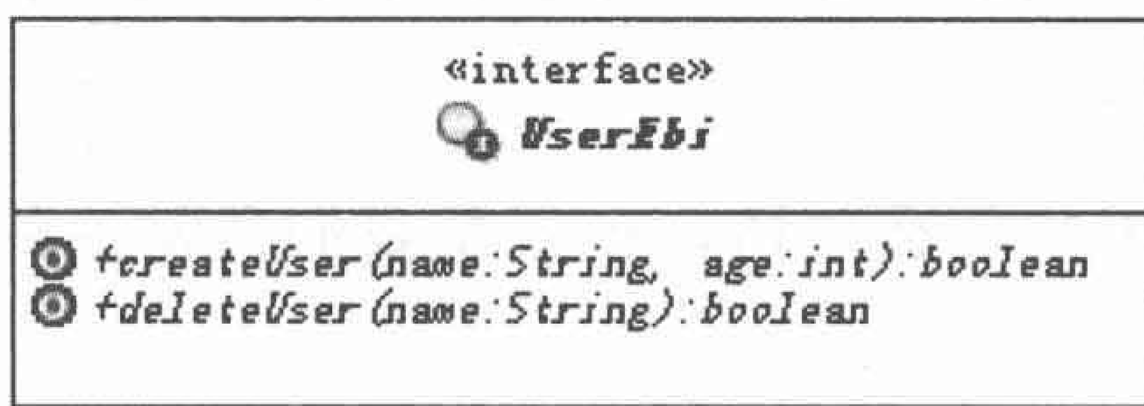


图 B.7　接口的类图

在其他 UML 图中，接口还可以用一个圆圈来表示，比如在组件图中。由于本书不涉及到这些知识，这里就不去示例了。

B. 2. 4　关系

前面讲到，类图除了描述类本身之外，另外一个重点就是描述类与类之间的关系。在 UML 2.0 中，类图描述的关系包括关联、泛化（也叫通用化或者继承）、依赖、实现、使用和流几种。大致如表 B.2 所示。

表 B. 2　关系的种类

关 系	功 能	表 达 形 式
关联	类实例之间的连接	标准的表达是一条直线：
泛化	概括描述和具体种类间的关系，适用于继承	
依赖	两个模型元素之间的关系	
实现	抽象说明和具体实现间的关系，如实现接口	
使用	一个元素需要使用其他元素功能的关系	
流	在相继时间内一个对象的两种形式的关系	

1. 关联关系

UML 的关联用于描述类和类的连接。类与类之间有多种连接方式，每种连接的含义都是不同的，虽然语义不同，但是外部表象类似，因此统称为关联。

关联关系一般都是双向的，也即关联双方都能和对方通信，但是也有单向的关联。根据不同的含义，把关联分成普通关联、递归关联、限定关联、或关联、有序关联、三元关联和聚合七种。本书涉及到了普通关联、递归关联和聚合几种，下面分别来介绍一下这三种关联，其他的关联关系就不再讲述了。

（1）普通关联

只要类与类之间存在关联关系就可以用普通关联来表示。标准的表示是一条直线，但是本书采用的是更严格的表达，如，带叉的这一端，表示关联的发起方，在另一端还可以通过一个箭头来表示被关联的一方，从而表示关联的方向，如。

来看看普通关联的示例：描述人和计算机的关系，如图 B.8 所示。

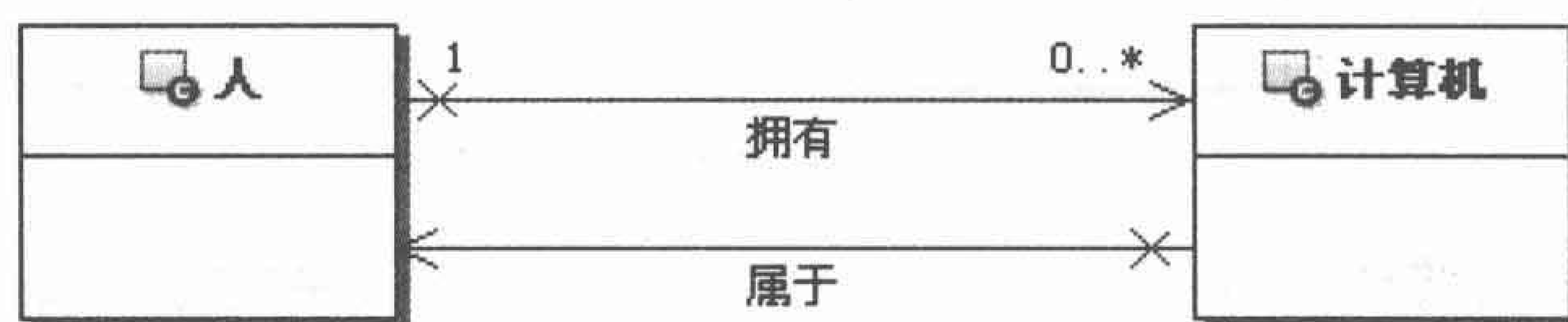


图 B.8 普通关联示例

图 B.8 中的数字表示重数，比如上面从人到计算机的关联，描述的是一个人拥有 0 到多个计算机。而另外一条从计算机到人的关联，没有重数，默认都是 1，表示一台计算机属于一个人。重数表示示例如下。

0..1	表示	零到 1 个对象
0..*或*	表示	零到多个对象
5..8	表示	5 到 8 个对象
2	表示	2 个对象
没有标示	表示	1 个对象

图 B.8 所对应的 Java 示例代码如下：

```
public class 人 {
    private Collection<计算机> col;
}
public class 计算机 {
    private 人 a;
}
```

这也是 Java 中表达对象之间一对多关系的方式，也就是在一的这边，包含一个多的那边对象的集合，而多的这边，包含一个一的那边的对象。

（2）递归关联

如果一个类与它本身有关联关系，那么这种关联关系被称为递归关联。

递归关联描述的是同类的对象之间的语义关系，带有递归的含义，通常情况下都是一对多的关系，如图 B.9 所示。

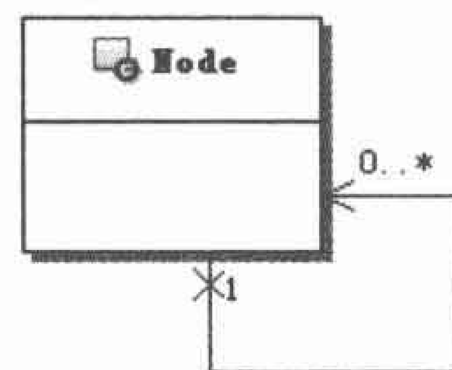


图 B.9 递归关联示例

当然递归关联也可以是一对一，或者多对多的，这里就不再多讲了。

（3）聚合关联

聚合关联是关联的一种特殊情况，如果类与类之间具有“整体与部分”的关系，使用聚合来表达。

根据语义又把聚合关联分成了三种：普通聚合、共享聚合和复合聚合（也叫组成）。

- 普通聚合：描述类与类之间具有“整体与部分”的关系。比如班级和学生，班级是整体，而学生是组成班级的部分，如图 B.10 所示。

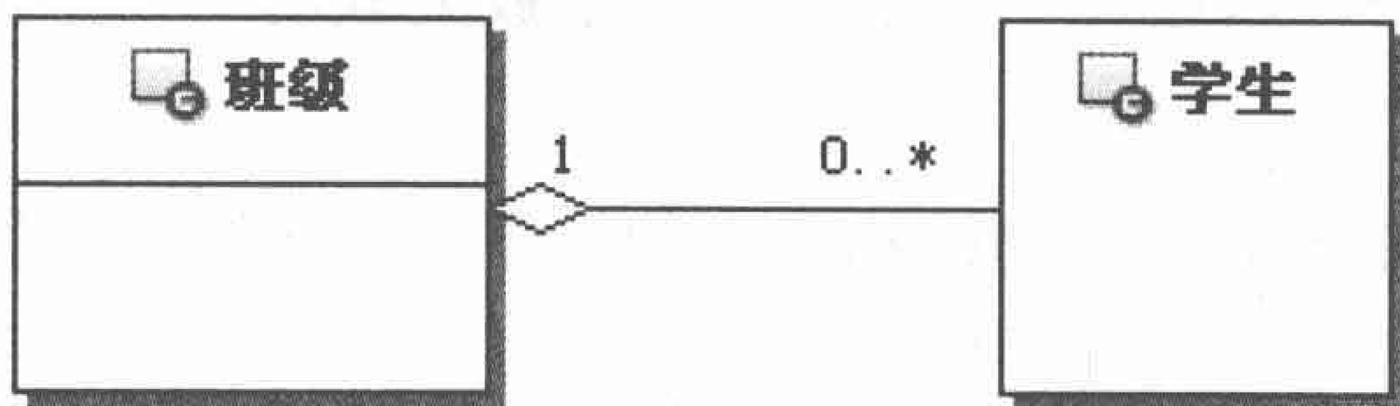


图 B.10　普通聚合的示例一

可以看出普通聚合的表达方式为，在表示关联关系的这端加上一个空心的菱形，空心的菱形紧靠着具有整体含义的这端，另一端可以没有标示，也可以加上箭头来表示一个方向性，如图 B.11 所示。

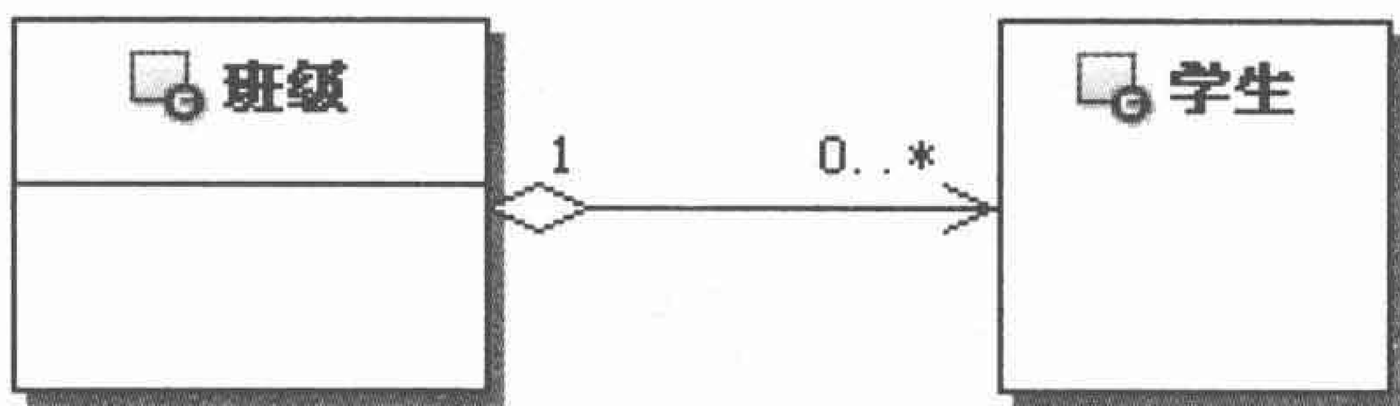


图 B.11　普通聚合的示例二

- 共享聚合：如果聚合关系中，处于部分方的对象参与了多个整体方对象的构成，描述成为共享聚合。比如学习兴趣小组和学生，学习兴趣小组是整体，而学生是组成学习兴趣小组的部分，但是一个学生可以参加多个学习兴趣小组，一个学习兴趣小组有多个学生，如图 B.12 所示。

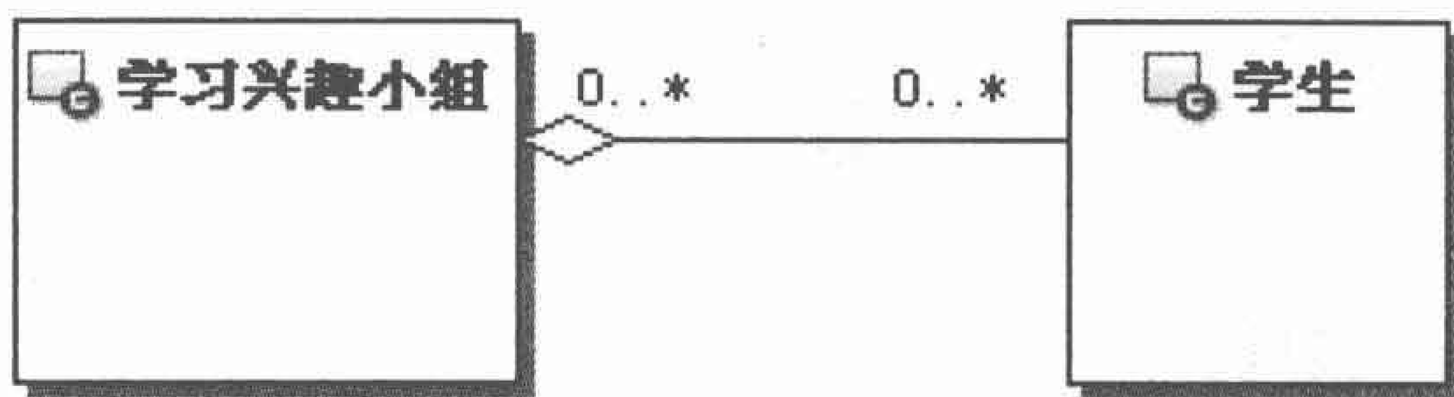


图 B.12　共享聚合的示例

可以看出在普通聚合中，整体与部分是一对多，而共享聚合中是多对多了。和普通聚合一样，可以在部分这边加上箭头，就不再示例了。

- 复合聚合：如果构成整体类的部分类，完全隶属于整体类，那么这样的聚合称为复合聚合，也叫组成。比如一个图形界面和组成这个图形界面的按钮、文本框、Label 等图形组件之间的关系，图形界面是整体，而各个图形组件是组成界面的部分，如图 B.13 所示。

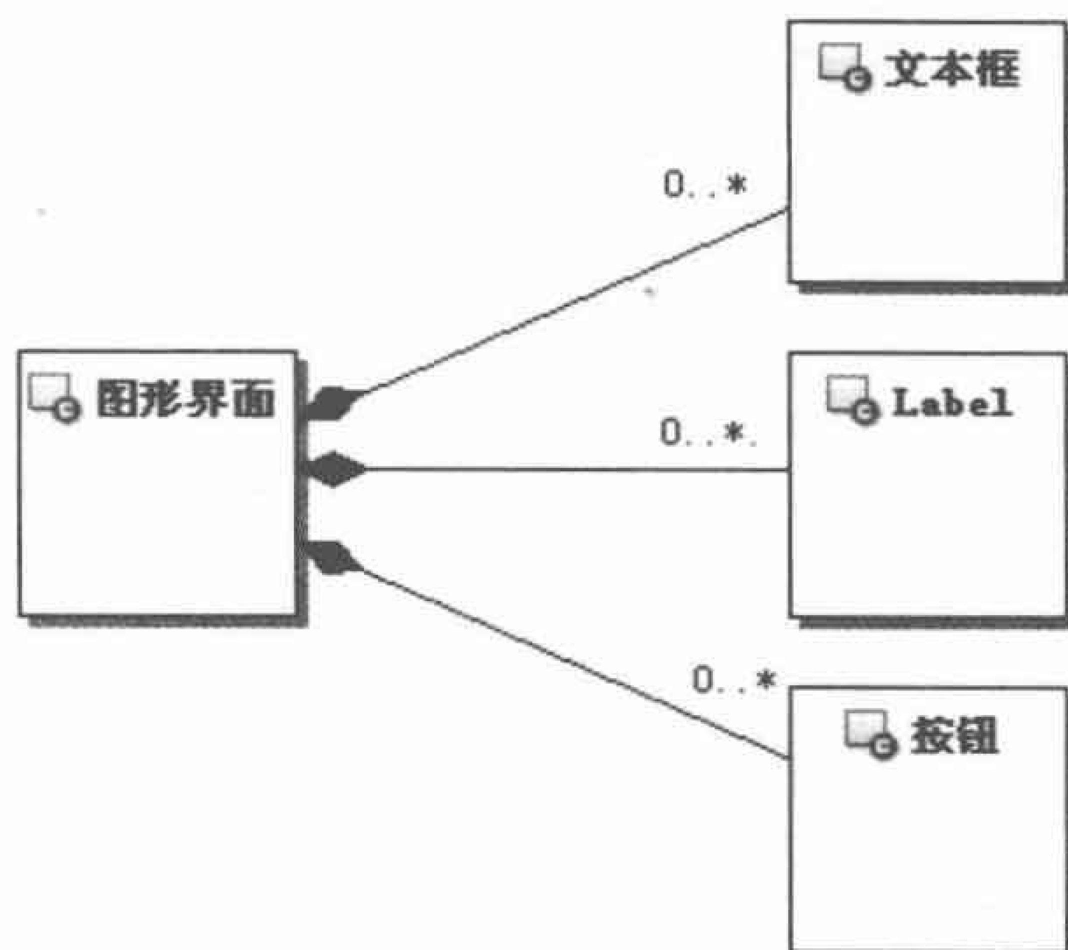


图 B.13 复合聚合的示例

可以看出复合聚合也是一对多的，只是在一的这端，用一个实心的菱形来表示了。通常复合聚合表达的语义有这样的含义：如果整体对象不存在，那么部分对象也就没有存在的前提或意义了，也就是说整体与部分有非常强烈的包含关系。和普通聚合一样，可以在部分这边加上箭头，就不再示例了。

2. 泛化关系

泛化又称通用化或继承，用来描述一个通用元素的所有信息能被另外一个具体元素继承的机制。继承某个类的类除了有自己的属性和操作外，还拥有被继承类中的信息。

比如人员和学生，明显人员是父类，学生作为子类，如图 B.14 所示。

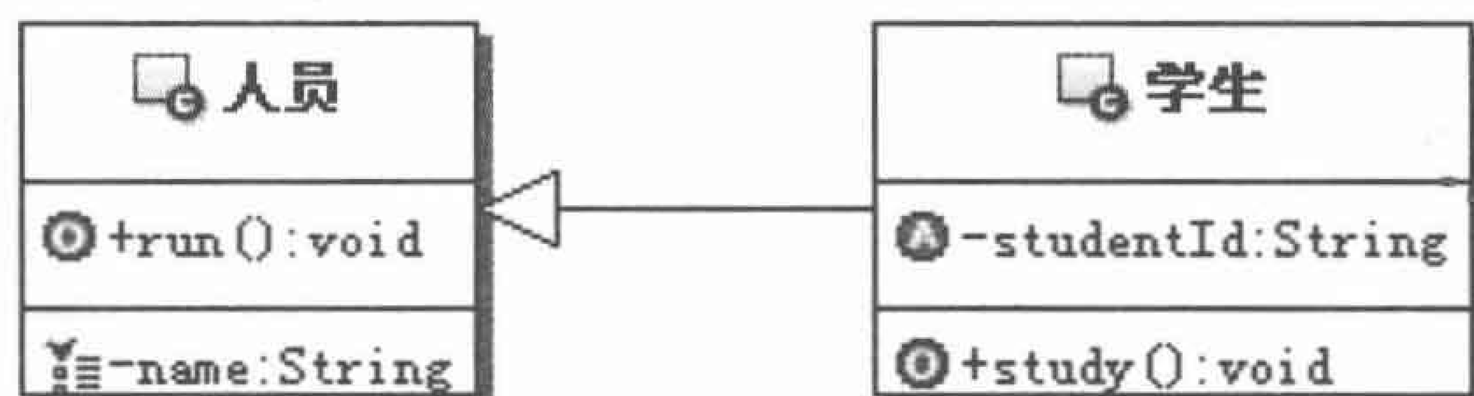


图 B.14 泛化关系的示例

3. 实现关系

就是描述类实现接口的关系。接口是对行为而非实现的说明，实现类来具体实现接口中的抽象定义。

比如，写一个实现前面定义的用户操作接口，如图 B.15 所示。

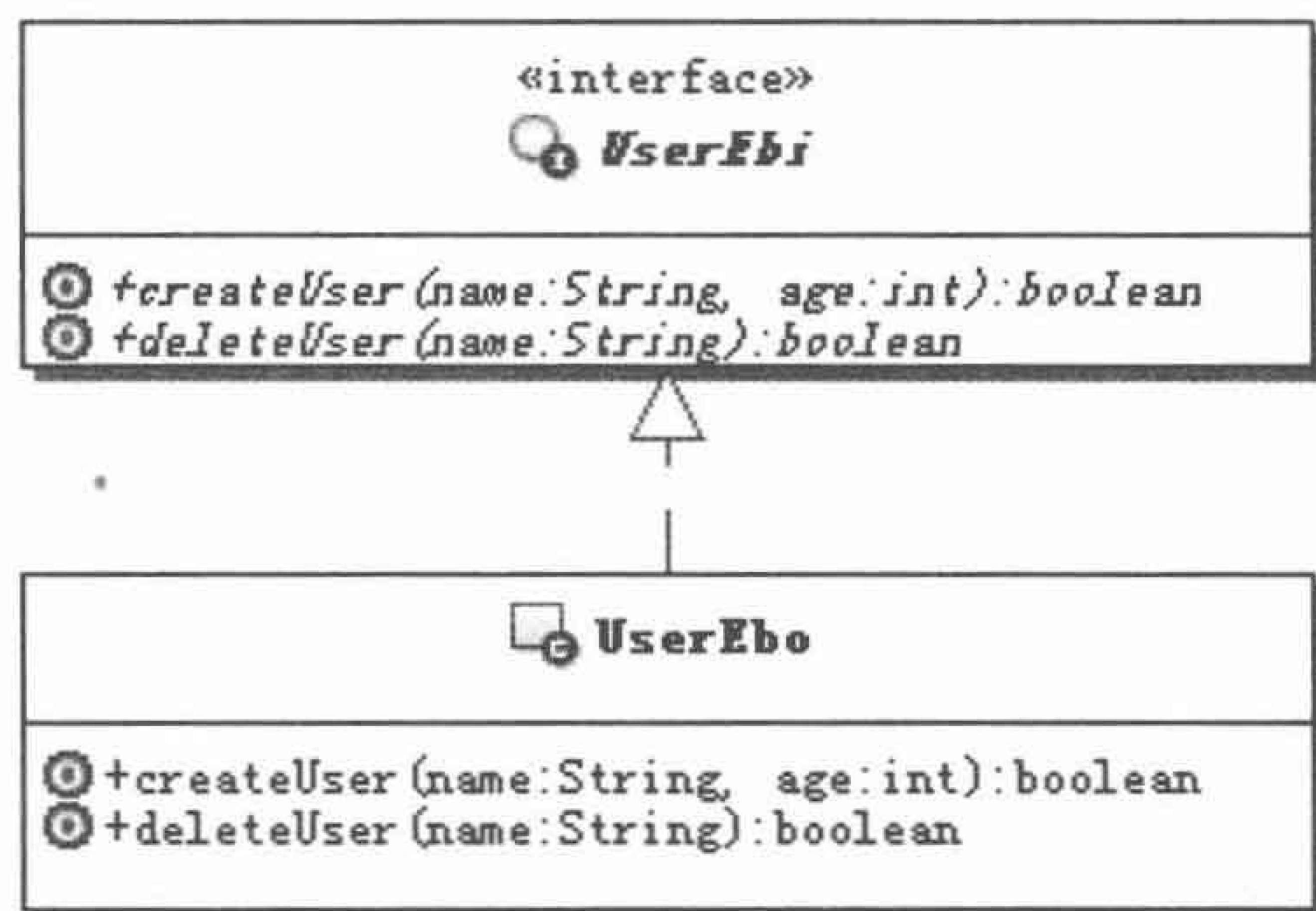


图 B.15 实现关系的示例

4. 依赖关系

依赖关系是描述：如果某个对象的行为和实现，需要受到另外对象的影响，那么就

说这个对象依赖于其他对象。基本上有关联的地方，严格说都有依赖。现在最常用的依赖关系是“使用”，意思是如果 A 使用了 B，那么 A 就依赖于 B。

比如，写一个 Client 来使用上面定义的接口和实现，那么这个 Client 就会依赖接口和实现这两个类，如图 B.16 所示。

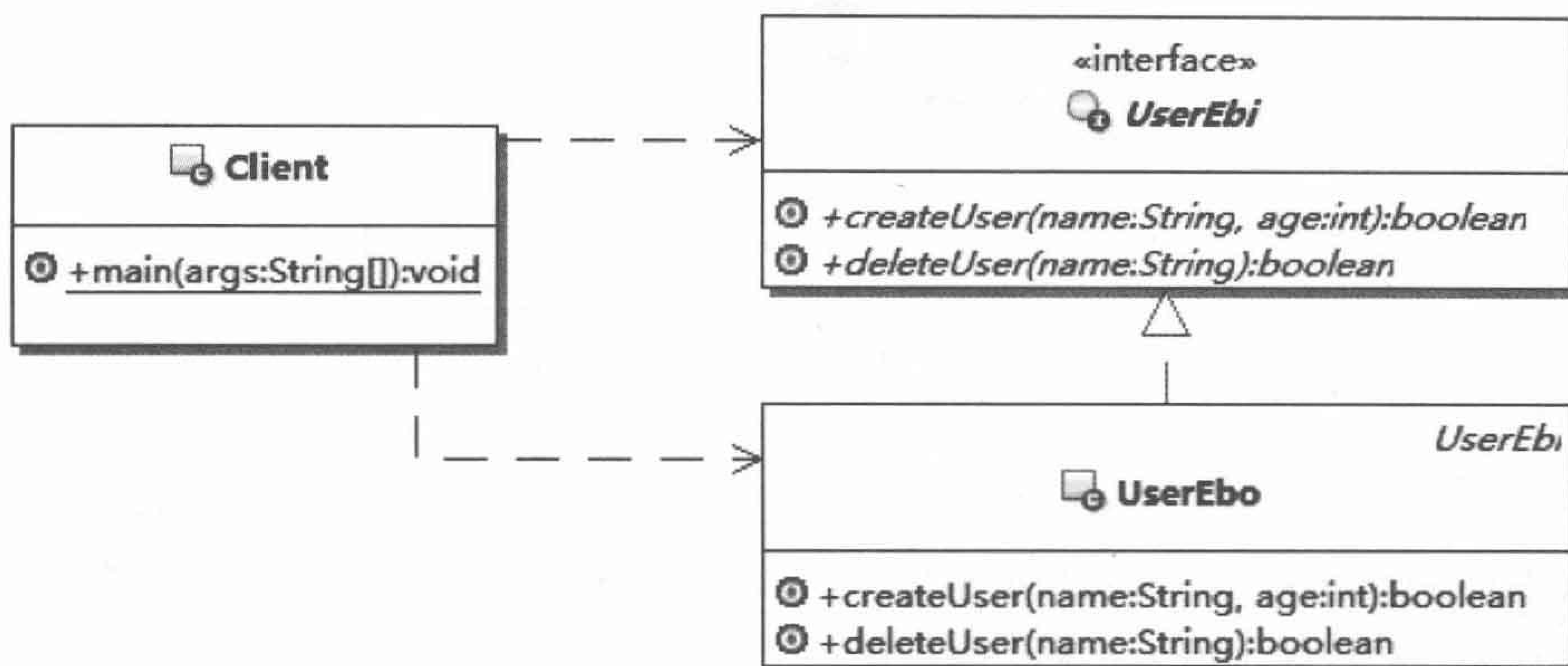


图 B.16　依赖关系的示例

Client 中的代码示例如下：

```
public class Client {
    public static void main(String[] args) {
        UserEbi ebi = new UserEbo();
        ebi.createUser("Test", 20);
    }
}
```

事实上，依赖关系描述的是两个模型元素之间语义上的连接关系。常见的除了上面的直接调用外，还有，一个类使用另外一个类来作为操作的参数，一个类有以另外一个类作为类型的属性，甚至包括上面讲到的继承和实现等关系，都可以算是依赖。

B.3　顺序图

B.3.1　顺序图的概念

顺序图是按照时间的先后顺序来描述消息是如何在对象间发送和接收的。顺序图有两个坐标轴，横坐标表示对象，纵坐标表示时间。顺序图又称为序列图或时序图。

顺序图中的对象用一个带有纵向虚线的矩形块来表示，矩形块中写有对象或类的名字，纵向的虚线表示对象的“生命线”。对象之间的交互用对象间的水平消息线来描述，消息线的箭头表示消息的类型，消息的先后顺序就是对象交互的先后顺序。

顺序图主要用来表示用例中行为的顺序，可以通过它来进行一个场景说明，或者是对某一个复杂业务功能或是流程执行先后的说明。也就是说，顺序图通常可以用来描述系统某次调用的运行顺序。

B.3.2 顺序图的基本表达

消息的表示方式如表 B.3 所示。

表 B.3 消息的表示方式

消 息 类 型	表 达 方 式
顺序消息	——>
异步消息	——⇀
调用	——▶

直接用例子来说明顺序图的基本表达方式，假若有 A 类和 B 类，A 类会调用 B 类的方法，然后写个客户端来调用 A 类，现在要使用顺序图来画出客户端的调用顺序。

假设 B 类的示例代码如下：

```
public class B {
    public int b1(int a){
        return a+1;
    }
    public void b2(String message,int sum){
        System.out.println(message+"="+sum);
    }
}
```

A 类的示例代码如下：

```
public class A {
    public void a1(int a){
        int sum = 0;
        B b = new B();
        for(int i=0;i<a;i++){
            sum += b.b1(i);
        }
        b.b2("求和的结果是", sum);
    }
}
```

Client 的示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        A a = new A();
        a.a1(5);
    }
}
```

1. UML 2.0 的顺序图示例

先来看看在 UML 2.0 当中如何画出这个顺序图，如图 B.17 所示。

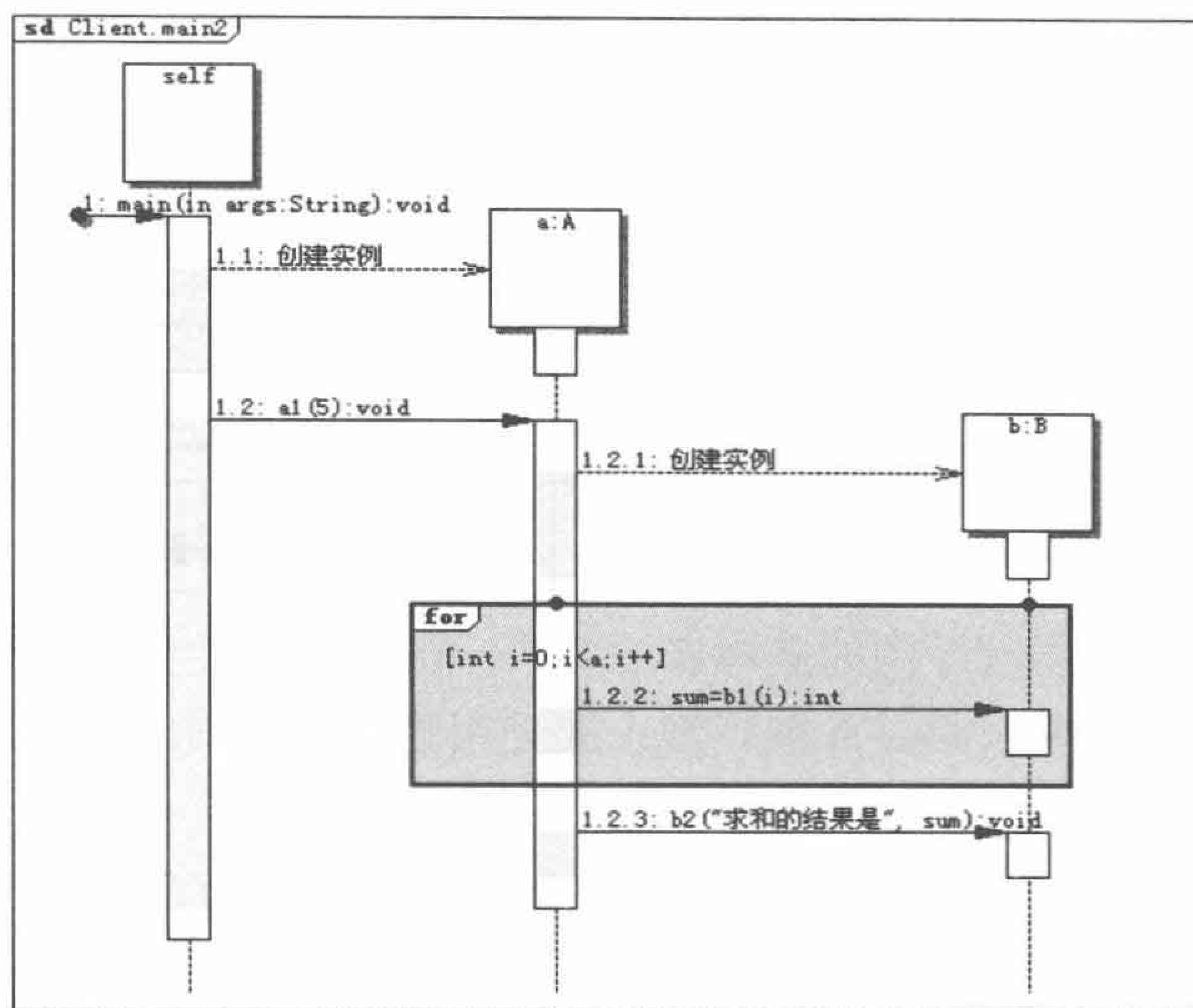


图 B.17　UML 2.0 顺序图示例

对图 B.17 稍微说明一下。

- 最外面那个大框框，在 UML 2.0 中称为 interaction，就是表示一个交互的这么一个边界，左上角的 sd Client.main2 是它的名字，是可以修改的。通常情况下，一个 interaction 应该和一个类中的某一个方法对应，相当于调用的起点，图 B.17 表示的就是从 Client 类的 main 方法开始的顺序图，直白点说就是当 Client 的 main 方法被执行或调用的时候，系统地调用顺序。
- 带虚线的方块就代表对象，self 就是表示 interaction 自身，也就是入口，即 Client 类的对象。
- 标号为 1 的那个从小黑点到对象 self 的一个调用，就是表示从外部来调用 Client 类，上面写着调用的方法是 main，这条横线加箭头，就表示消息。
- 虚线上的矩形条表示对象被激活的生命周期，简单点说，就是在这一段对象被其他对象使用。
- 那个 for 的块，在 UML 2.0 中表示 Combined Fragment（组合片段），也就是用来进行块控制的，比如 if-else、for、while、try 等。

大家看到这个顺序图的示例，会发现和前面各个模式中画的顺序示意图有很大的不同，的确是这样的。因为 UML 2.0 的顺序图的语义丰富了不少，当然表达的越细致，图就会越复杂。因此，前面每个模式的运行顺序示意图，是采用的 UML1.4 的版本来绘制的。因为 UML 1.4 相对而言更简单一些，而且我们要表达的也比较粗略，只是想要描述出模式运行起来的大概顺序，并不深入细节。

2. UML1.4 的顺序图示例

使用 UML 1.4 画出上面示例的顺序图，如图 B.18 所示。

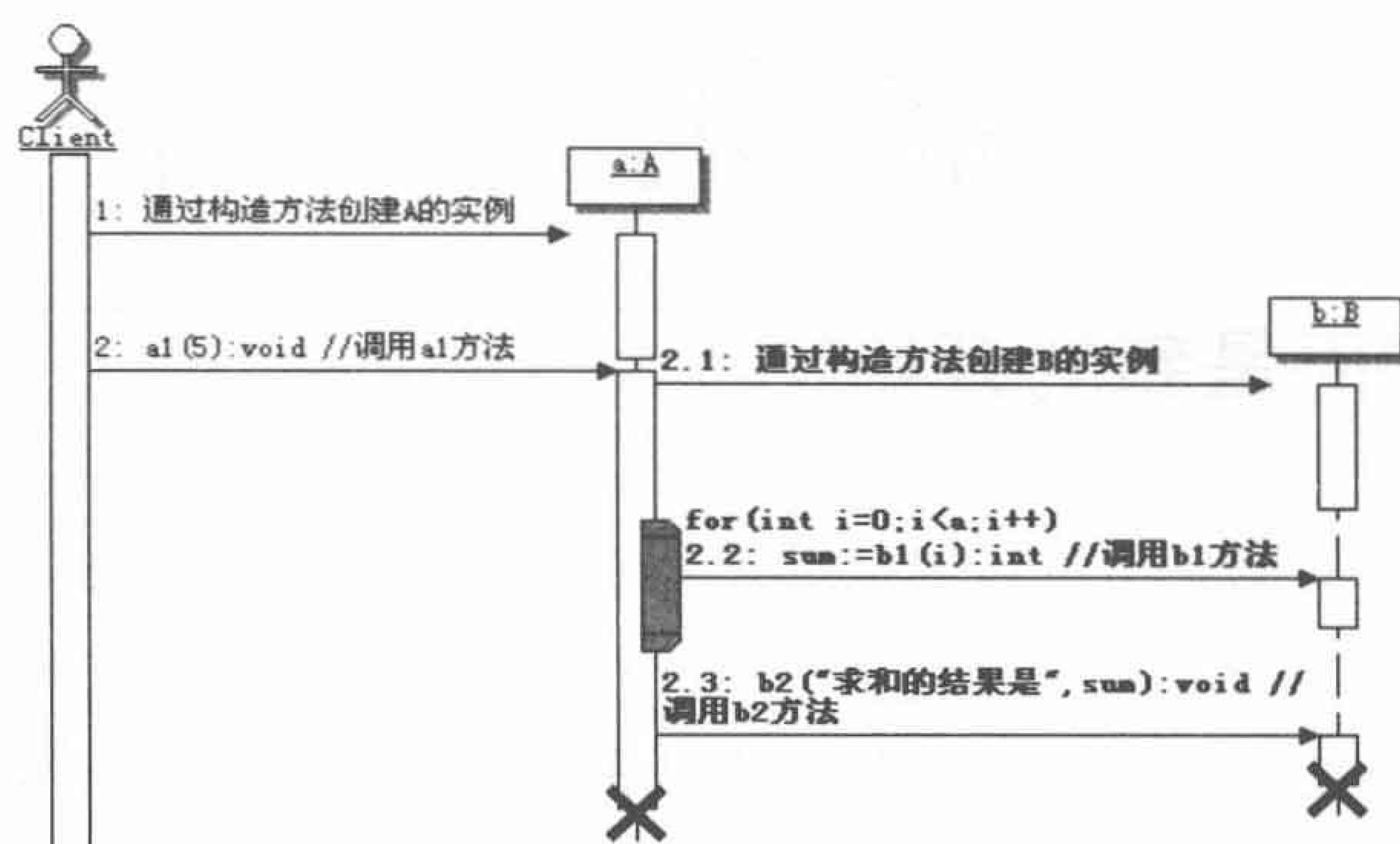


图 B.18　UML 1.4 顺序图示例图一

图 B.18 生命周期下面的大黑叉叉，表示到这个地方销毁本对象实例。

这个图大家看起来可能要简单很多，至于后面的 A 和 B 对象画得比 Client 对象矮，是为了表示出运行到某个时间才创建的对象，当然也可以把它们放到一个水平线上。把上面的顺序图调整一下，可以画出如图 B.19 所示的顺序图。

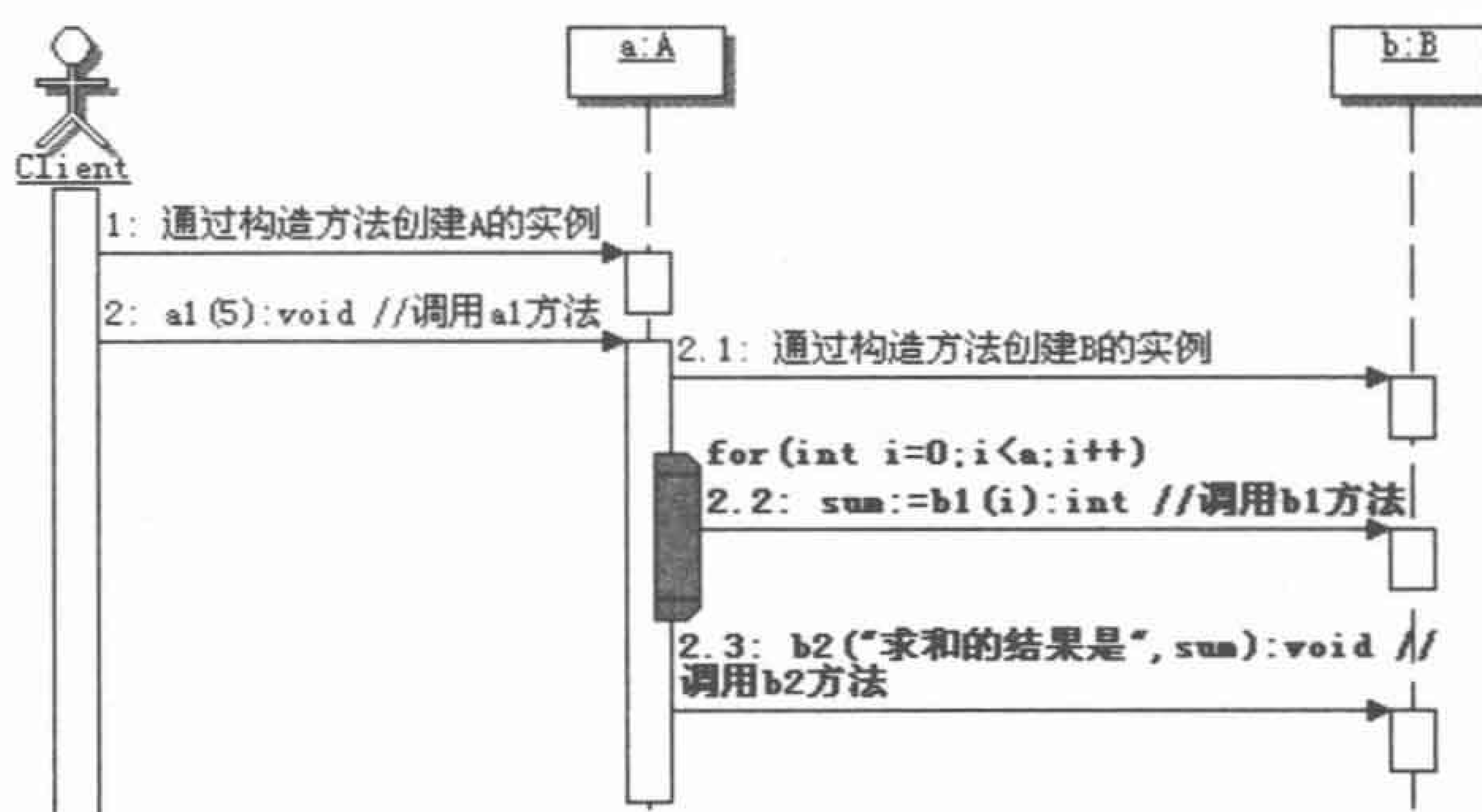


图 B.19　UML 1.4 顺序图示例图二

图 B.19 基本上就是前面画每个模式的运行顺序示意图的大概样子。由于前面画每个模式的运行顺序图的时候，很多都是文字说明，并没有和真实的代码结合，所以都没有直接称为顺序图，而是称呼为顺序示意图，表示不是完整的顺序图，这点大家要了解一下。好在我们是把重点放在对模式运行顺序的理解上，并不是来讲述 UML 图如何画，所以不必把它们当严格的、完整的 UML 顺序图来看待。

好了，对于 UML 的基础知识，这里就简略地介绍这么多，了解这些知识，对于看懂本书所绘制的 UML 图应该是足够了，如果想要了解更多的 UML 的知识，请参考专业的 UML 方面的资料。

临别赠言

不是结束而是新的开始

首先恭喜你，看到这里，说明你已经基本掌握了本书所讲述的设计模式的内容，应该可以达到中级水平了。

但是，这并不是说你就不用再学习设计模式了，恰恰相反，要想在设计上更进一步的话，困难才刚刚开始。从中级的水平向上发展，更多的是需要思考和领悟，其难度比从入门到中级要大得多。

因此对你而言，看完本书并不是学习的结束，而是新的开始。

你该怎么做

如果已经深入领会和掌握了本书的内容，还想要在设计模式上继续精进的朋友，给出如下的建议。

1. 多看

多搜寻一些应用设计模式的实际项目、工程或是框架，参考别人的成功应用，再结合自己的经验来思考和使用。当然项目不应该太大，太大了很难完全看懂；也不能太小，太小了，没有太大实用价值，尤其是无法参考多个模式综合应用的情况，帮助就不大了。

2. 多练

多寻找机会，把这些设计模式在实际应用中使用，只有亲自动手去试验和使用，才能真正掌握和领会设计模式的精髓。

3. 多总结

认真分析每次对设计模式的使用是否得当，有什么经验和教训，是否有变形使用的情况，在不断总结中进步。

4. 反复参阅本书

理论联系实际，通过实际应用反过来加深对理论的理解，以达到融会贯通这些设计模式的知识。因此，你需要反复参阅本书，看看书上的知识，然后实践，再回头看书上的知识，你会有不一样的体会和领悟。

5. 多思考

多从设计上去思考这些设计模式，考虑它的设计意图、设计思想、解决问题的方式、实现的原理、模式的本质，以及如何变形使用等。

只要你坚持按照上面说的做，假以时日，必有所成。**预祝大家成功！**

参 考 文 献

资料名称：《设计模式——可复用面向对象软件的基础》

作　　者：Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 著

李英军、马晓星、蔡敏、刘建中等译

出 版 社：机械工业出版社

出版年份：2000.9

资料名称：《设计模式 Java 手册》

作　　者：Steven Hohn Metsker 著

龚波、冯军、程群梅 等译

出 版 社：机械工业出版社

出版年份：2006.6

资料名称：《Effective Java 中文版》

作　　者：Joshua Bloch 著

杨春花 俞黎敏 译

出 版 社：机械工业出版社

出版年份：2009.4

资料名称：《J2EE 核心模式》

作　　者：Deepak Alur、John Crupi、Dan Malks 著

刘志奇、丁天、田蕴哲 等译

出 版 社：机械工业出版社

出版年份：2002.1

资料名称：《敏捷软件开发原则、模式与实践》

作　　者：Robert C.Martin 著

邓辉 译

出 版 社：清华大学出版社

出版年份：2003.9

资料名称：Design Patterns In Java

作　　者：Bob Tarr

资料来源：网络免费资源，电子 PPT

资料名称：《UML2 参考手册》

资料来源：网络免费资源，网址是：http://china.pub.com

参考文献

资料名称：《设计模式——可复用面向对象软件的基础》
作　　者：Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 著
　　　　　李英军、马晓星、蔡敏、刘建中等译
出 版 社：机械工业出版社
出版年份：2000.9
资料名称：《设计模式 Java 手册》
作　　者：Steven Hohn Metsker 著
　　　　　龚波、冯军、程群梅 等译
出 版 社：机械工业出版社
出版年份：2006.6
资料名称：《Effective Java 中文版》
作　　者：Joshua Bloch 著
　　　　　杨春花 俞黎敏 译
出 版 社：机械工业出版社
出版年份：2009.4
资料名称：《J2EE 核心模式》
作　　者：Deepak Alur、John Crupi、Dan Malks 著
　　　　　刘志奇、丁天、田蕴哲 等译
出 版 社：机械工业出版社
出版年份：2002.1
资料名称：《敏捷软件开发原则、模式与实践》
作　　者：Robert C.Martin 著
　　　　　邓辉 译
出 版 社：清华大学出版社
出版年份：2003.9
资料名称：Design Patterns In Java
作　　者：Bob Tarr
资料来源：网络免费资源，电子 PPT
资料名称：《UML2 参考手册》
资料来源：网络免费资源，网址是：http://china.pub.com